TR 87-46

CSP-i: AN IMPLEMENTATION OF CSP

Thesis submitted by

KAREN LEE WRENCH

In Fulfillment of the
Requirements for the degree

MASTER OF SCIENCE

Rhodes University
November 1986

ACKNOWLEDGEMENTS

I wish to express my sincere thanks to my supervisor and the originator of this project, Peter Clayton, for his advice, encouragement and constructive criticism.

Thanks to Mike for the much needed laughter in the office, and to John for contributing to my sanity, or rather insanity?

Words of love to my parents, for their unceasing support throughout my extended university career.

I would also like to acknowledge the financial assistance of Rhodes University and the Council for Scientific and Industrial Research.

ABSTRACT

CSP (Communicating Sequential Processes) is a notation proposed by Hoare, for expressing process communication and synchronization. Although this notation has been widely acclaimed, Hoare himself never implemented it as a computer language. He did however produce the necessary correctness proofs and subsequently the notation has been adopted (in various guises) by the designers of other concurrent languages such as Ada and occam.

Only two attempts have been made at a direct and precise implementation of CSP. With closer scrutiny, even these implementations are found to deviate from the specifications expounded by Hoare, and in so doing restrict the original proposal.

This thesis comprises two main sections. The first of these includes a brief look at the primitives of concurrent programming, followed by a comparative study of the existing adaptations of CSP and other message passing languages. The latter section is devoted to a description of the author's attempt at an original implementation of the notation. The result of this attempt is the creation of the CSP-i language and a suitable environment for executing CSP-i programs on an IBM PC.

The CSP-i implementation is comparable with other concurrent systems presently available. In some aspects, the primitives featured in CSP-i provide the user with a more efficient and concise notation for expressing concurrent algorithms than several other message-based languages, notably occam.

CONTENTS

page

1. INTRODUCTION 1

2. CONCURRENT PROGRAMMING CONCEPTS

2.1. Processes 4
2.2. Process Interaction 5
2.3. Communication Mechanisms 6
2.4. Synchronization using Shared Variables 7
2.5. Message Based Synchronization Systems 13

3. CONCURRENT LANGUAGES

3.1. Communicating Sequential Processes 21
3.2. Distributed Processes 23
3.3. Ada 26
3.4. Occam 28
3.5. Synchronizing Resources 29
3.6. Communication Ports 30
3.7. PLITS 31
3.8. Some General Observations of Message Based Languages 32

4. AN IMPLEMENTATION OF CSP

4.1. Original Definition of the Notation 34
4.2. Early Implementations 35
4.3. A New Grammar for CSP 37
4.4. Deviations from CSP in the Syntax of CSP-i ... 39

5. THE CSP-i COMPILER

5.1. Choice of Parsing Method 41
5.2. Declarations and the Use of the Symbol Table 43
5.3. Parsing Selected CSP-i Constructs 51
5.4. Suspension of Code Generation 61
5.5. An Alternative Implementation for CSP-i ... 62

6. THE CSP-i INTERPRETER

6.1. The Preprocessor 64
6.2. The Scheduling Procedure 66
6.3. The Intermediate Language Code Interpreter ... 67

7. REFINEMENTS AND EXTENSIONS TO CSP

7.1. Clean Termination of Processes 80
7.2. Implementation of a Generalized Input/Output Command 82
7.3. An Otherwise Option for the Alternative Command 95

8. CRITICAL EVALUATION AND CONCLUSION

8.1. The Basic CSP-i System 98
8.2. Directions for Future Research 103

BIBLIOGRAPHY

APPENDICES

A. CSP-i Syntax
B. CSP-i Intermediate Language Codes
C. Sample CSP-i Programs
D. Program Listing - Compiler
E. Program Listing - Interpreter

Introduction

1. INTRODUCTION

In 1978 Hoare published a proposal for a new programming notation based on the underlying assumption that input, output and concurrency should be regarded as primitives of programming [Hoa78]. Although these primitives can by no means replace the familiar concepts used in high level languages such as assignment, they may be used in conjunction with these more established primitives to design a language suitable for both shared memory and distributed architectures.

The Communicating Sequential Process notation (commonly referred to as CSP) was an initial attempt to provide the basis for the development of such a concurrent language.

Although the design of several high level languages has been influenced by this new concept, and although mathematical proofs exist which confirm its suitability for concurrent algorithms, very few language implementations have accurately reflected the syntax and design goals of the CSP notation. Challenged by the absence of authentic CSP implementations, the author set about the task of implementing CSP as a problem solving language on a shared memory system.

A new language, CSP-implemented (hereafter referred to as CSP-i) is described in this thesis. The structure of several other concurrent languages is also studied with the view to deducing the extent of CSP's influence on these notations and surveying the alternative message passing schemes currently in use.

In this thesis, it has been assumed that the reader is familiar with the principles of concurrent programming and has a basic knowledge of the CSP notation. The syntax of CSP as such, and the construction of concurrent programs using this notation will not be discussed. The author recommends the paper by Hoare [Hoa78] for a general

introduction, and his recent book [Hoa85] as a more comprehensive guide to CSP.

The research undertaken is documented as follows:

Chapter 2 acts as an introduction to the concurrent programming concepts of processes and process interaction using communication and synchronization primitives. In particular, the primitives which rely on shared memory and those best suited to distributed systems are distinguished and a brief history of each is given. The alternatives available in designing a message passing scheme are also considered.

In Chapter 3, we concentrate on those concurrent languages which use message passing to effect interaction between processes. In the description of each language, only those features of the language directly concerned with interprocess communication are dealt with. The chapter concludes with a few general comments on the advantages and disadvantages of the various notations discussed.

The definition of CSP and known implementations of the notation are discussed in Chapter 4. CSP-i is introduced and compared to the original CSP notation.

Chapters 5 and 6 describe the CSP-i compiler and interpreter respectively. Once again emphasis is placed on the implementation of the novel constructs of the language, notably the use of messages, input/output commands, the guarded command and the repetitive command. An alternative approach to the one taken by the author in the implementation of CSP-i, is presented at the end of chapter 5.

Chapter 7 investigates some of the extensions proposed in the literature, with emphasis on the implementation of a generalized input/output command and the provision for the termination of processes. The incorporation of these features in the CSP-i implementation is fully documented.

The thesis is concluded with a critical evaluation of the CSP-i implementation and an appraisal of the project as a whole.

Included as appendices are

- the full CSP-i syntax in the form of production rules,
- a list of the intermediate language codes used as the interface between the compiler and interpreter,
- sample CSP-i programs, and
- program listings for the compiler and interpreter.

Concurrent Programming Concepts

2. CONCURRENT PROGRAMMING CONCEPTS

2.1. Processes

The very first computer programming languages were designed as formal notations for expressing sequential programs. A sequential program consists of a sequence of commands or statements which are executed individually and in some sequence directly determined by the semantics of the statements. The execution of such a sequential program is called a process.

In the past decade, much research has centered on the development of concurrent programming. Two significant factors have contributed to the great interest and rapid progress in this field.

- The sequential execution of a series of actions is alien to the human mind. We are far more inclined to and capable of doing a variety of disjoint actions concurrently. In the development of a more human-like computer language, it would therefore be necessary to include a similar means of expressing the parallel execution of commands.

- Advances in technology have made available inexpensive processors, thus enabling the construction of distributed systems and multiprocessors, which were previously infeasible. These new architectures provide the hardware for the direct execution of concurrent programs.

Concurrent programs can be defined as two or more sequential programs which are executed in parallel with one another. This gives rise to the term parallel processes.

An automated banking system where transactions can be processed on any one of the automatic teller machines (ATM's) in the system, has a natural specification as a concurrent program. Each ATM can be

constructed as an individual process, controlled by a sequential program. In order for the system to operate efficiently though, these individual processes must be activated simultaneously.

Besides providing a means whereby several real-time computer applications can be expressed using concurrent algorithms, parallel processes are more time-efficient than their sequential counterparts.

By allowing various components of a program to be executed simultaneously on different processors, the time taken for the execution of the program as a whole can be remarkably reduced. Even processes multiprogrammed on a single processor are more time efficient than an equivalent sequential program, since lengthy input/output sequences in one process can be done in parallel with the utilization of the CPU by another process.

2.2. Process Interaction

Concurrent programming systems are of limited use if there is no co-operation between the parallel processes constituting the program. This means that some form of synchronization and communication between parallel processes must be provided to enable them to use shared resources effectively, pass data from one process to the other and generally control the execution speed of one process in relation to the progress of another process.

Language proposals for concurrent systems usually define a basic component, which is similar to that provided in sequential language systems, plus additional facilities for external cooperation.

Several new notations have been designed and some existing high level languages extended to incorporate these concurrent features. These include

Ada	[Geh84a, Geh84b, Brn82]
CHILL	[Fid83, Sme83]
Clang	[Cha84]
Concurrent Pascal	[Bri75]
CSP	[Hoa78, Hoa85]
Distributed Processes	[Bri78]
Edison	[Bri82]
Modula-2	[Wir83]
Occam	[May83, Jon85]
Pascal-Plus	[Bus80]

Most concurrent notations differ primarily in the primitives provided for specifying synchronization and communication. Various co-operation mechanisms are discussed in the remainder of this chapter.

2.3. Communication Mechanisms

As defined by Lauer [Wil84], communication mechanisms can be divided into two groups according to the architecture of the computer system on which the processes are executing;

- either a single processor or multi-processor with shared resources (mainly memory), known as a "procedure-oriented" system,
- or a distributed system with no shared memory, that is, a "message-oriented" system.

In shared memory systems, processes exchange data by reading from and writing to shared variables. Synchronization can be achieved by setting and testing these shared variables.

In distributed systems, various forms of explicit movement of data via message passing are available for communication between processes. It should be noted however, that a message passing scheme can also be implemented on an architecture naturally suited to a procedure-oriented system.

It is often difficult to distinguish between synchronization and communication issues in any particular language proposal since they can be implemented by the same primitive. In the discussion that follows, the author will concentrate on synchronization primitives, with additional comments on communication facilities where these are provided by different language features.

2.4. Synchronization using Shared Variables

Two types of synchronization are required on systems with shared resources, namely mutual exclusion and conditional synchronization [And83].

By mutual exclusion is meant that at any one time, only one process may be executing a certain sequence of instructions. Known as the critical region of the program, these instructions would normally manipulate shared resources. If the operations on these resources were not indivisible, the resulting output would be undefined and worthless.

Consider an airline reservation system, where updates and queries to a database can be made from several terminals at one time. Any update of the database should be completed before another transaction is allowed to commence further alterations. This is an example of the well known "readers and writers" problem described in [Ben82].

Conditional synchronization allows one process to be delayed until another process reaches a pre-determined point in its execution. This can be illustrated by the "producer-consumer" problem, where a consumer should be delayed until the producer has deposited at least one product in the buffer [Ben82].

Several methods of implementing these synchronization requirements are in general use today.

2.4.1. Busy waiting

Processes can synchronize by setting and testing shared variables. Using this mechanism, conditional synchronization can be implemented relatively simply. To signal a condition, a process sets the value of a variable. Another process waiting on the same condition must repeatedly test this variable until it has the desired value.

Mutual exclusion is more clumsy to implement, although Dekker and Peterson have presented solutions for its implementation [And83].

PL/1 was one of the first languages to provide limited concurrent features using shared variables and a busy-waiting mechanism to control access to these variables [Wil84]. On the whole however, a busy-waiting scheme is very wasteful of processor time. Furthermore, programs are difficult to follow since program variables are used both for implementing synchronization and for interprocess communication.

2.4.2. Semaphores

A semaphore is a non-negative integer variable on which two indivisible operations are defined, namely P (wait) and V (signal). P(s), where s is a semaphore, delays until $s > 0$ and then decrements s by one. V(s) can be executed at any time and increments the value of s by one.

Devised by Dijkstra [Bri73a], semaphores are a very general and efficient tool for implementing both types of synchronization. Mutual exclusion can be achieved by preceding each critical section of code with a wait operation on a binary semaphore, and concluding the section with a signal. With the binary semaphore initialised to one, only one process will be able to enter a critical region bounded by this semaphore.

Counting semaphores are used for conditional synchronization when

controlling shared resources. The value of the semaphore represents the number of resource still available. A queue is associated with the semaphore in order to prioritise the simultaneous requests by processes trying to gain access to the resource.

Several earlier concurrent programming languages incorporate semaphores, for example Algol68, Clang, Concurrent Pascal and Pascal-Plus. With the advent of distributed programming however, other mechanisms have been preferred, since semaphores were never intended to be used in programs executing in a distributed system. Implementing an operating system for such a scheme would be a non-trivial task, as a list of blocked processes (that is, those waiting on semaphores) must be available to all processors to enable the efficient allocation of processes to processors.

Generally semaphores do not have the higher level of abstraction required to model distributed communication and synchronization.

2.4.3. Critical regions

Critical regions, devised by Brinch Hansen [Bri73a], provide a more structured synchronization primitive.

Synchronization and communication mechanisms are disjoint, with the former implemented by the use of so-called critical regions and the latter, process communication, via shared variables.

The critical region construct is syntactically equivalent to a P and V pair used to implement mutual exclusion using semaphores. Once a resource R has been declared to consist of several shared variables, access to these variables can only take place within a critical region naming R. Using the notation in [Bri83a], a resource of type T would be declared as

```
var R: shared T;
```

Changes to any of the shared variables contained in R can only take place within a region statement of the form

region R do <statement list> end

Mutual exclusion is guaranteed by virtue of the definition of critical regions, namely that only one region statement involving a particular resource can be executed at any time, and by restricting the declaration of shared variables to one resource only.

Conditional critical regions are an extended form of critical regions. Incorporating a "when clause", they provide conditional synchronization. Using the notation proposed by Hoare [Bri82], a conditional region statement can be expressed as

with R when size < max do <statement list>

As the Boolean condition in a conditional clause is determined by repeated testing, and consequently uses busy-waiting, implementation of conditional critical regions on a single processor is costly.

Conditional critical regions have however, been used successfully in the Edison language, which was designed especially for a multiprocessor system [Bri82]. Regions as such, are not used in Edison; yet mutual exclusion is achieved by allowing only a single process to be executing the critical part of a **when** statement at a time.

Variants of this construct have also been used for distributed synchronization in the language proposal Distributed Processes [Bri78].

2.4.4. Monitors

In a language such as Edison, critical regions can be dispersed throughout the program, making it fairly awkward to see exactly what

operations are performed on each of the shared variables. The use of monitors [Bri73b], alleviates this problem since all shared variables plus the procedures defining the operations to be performed on them are encapsulated in one data structure, the monitor.

The monitor model defines two main constituents, namely processes which are active, and monitors which are passive. Processes enter the monitor by calling one of the monitor procedures. Since only one process may be executing within a particular monitor at a time, a queue is associated with each monitor to schedule entry of processes calling these monitor procedures.

Hoare [Hoa74] extended this monitor concept by allowing condition variables to be declared within a monitor. Two operations, wait and signal, are defined on these variables and are used in conjunction with each other to block a process executing inside a monitor. Such a blocked process is forced to relinquish exclusivity on the monitor. Additional queues are associated with these condition variables to enable blocked processes to regain exclusivity of the monitor once it becomes free and once the condition variable upon which the process was blocked, is found to be true.

Concurrent Pascal [Bri75] was one of the first languages to incorporate the monitor concept, with Modula [Wir77] and subsequently Modula-2 [Wir83] following shortly afterwards. Other languages using monitors to achieve communication and synchronization between processes are Pascal-Plus, Clang and CHILL.

Although none of the languages mentioned above were designed for execution in a distributed system, the local monitor concept can be extended to a distributed monitor concept by means of the remote procedure call. A language designed by Cook and known as *MOD, makes use of this extension to allow communication between nodes in a distributed system [Wil84].

2.4.5. Path expressions

In an attempt to localise all synchronization in one place, Campbell and Habermann [Lau78] introduced a new method of describing the communication of processes, namely path expressions.

Similar to the monitor concept, this notation combines both the explicit synchronization (the wait and signal monitor operations) and the implicit synchronization (mutually exclusive access to the monitor) into one statement. These synchronization statements are then associated with the processes using a resource. The "resource-oriented" nature of this notation, enables all resources to be viewed as entities which are capable of making their own decisions about the use of their constituent procedures by other processes [Lau78].

Each procedure whose execution is to be synchronized must be named in the path statement, together with the conditions for its synchronization. A process calling such a procedure may proceed so long as none of these synchronization conditions are contravened; otherwise the calling process is delayed.

As an example, the collection of paths to illustrate a read-write resource in the "readers and writers" problem could be expressed as

path write, $read_1$ end
path write, $read_2$ end
....
path write, $read_n$ end

The write operation appears in all paths, which means that each writer process has exclusive access to the resource. On the other hand, one or more reader processes may be activated concurrently.

While path expressions are particularly suited to describing operational synchronization (that is, the permissible sequence of execution of the processes manipulating a resource), they are not able

to describe conditional synchronization adequately. In the design of Concurrent C [Tsu84], this restriction was overcome by combining the use of control expressions and monitors for interprocessor communication.

2.5. Message Based Synchronization Systems

A message passing scheme to be used in a distributed or multiprocessor system must provide

- identification of the processes involved in the communication,
- the ability to transfer data between two processes, and
- synchronization of the processes' actions.

At the same time however, such a scheme must ensure and encourage a high degree of concurrency [Gen81].

Several variations of message based systems have been proposed and implemented. All of these include the basic facilities to receive and send messages. The differences in the proposals are due to the semantics attached to these send/receive primitives.

Communication in a message based system is provided by the actual transfer of data, while synchronization is ensured by enforcing the causality constraint whereby no message may be received before it has been sent. Gentleman [Gen81] identifies four issues which dominate the semantics of message passing. These are

- whether communicating processes block other processes during the sending and receiving of messages,

- what addressing scheme is used to identify the processes taking part in the communication,

- what constitutes a message and in what format it is transferred, and

- how failures in communications are handled.

These issues are discussed more comprehensively in the next four subsections and will be used as the basis of the comparison between implementations of message based languages in Chapter 3.

2.5.1. Asynchronous or synchronous communication

A receive or send statement is said to be blocking if its execution can cause a delay in the invoking process; otherwise it is a non-blocking primitive.

In using a blocking message-passing statement, no other facilities for synchronization need be provided. If the partner process in a communication is not ready to complete the transaction, the initiating process, which could be either the receiver or sender of the message, blocks until such a time as the named process reaches a point in its execution where it is able to communicate with the delayed process.

An advantage of this interpretation is that no message buffers are needed, since the sending process cannot actually transmit its message until the receiver is ready to accept it. Synchronous communication also implies that the sender cannot proceed arbitrarily far ahead of the receiving process.

Blocking message passing tends to inhibit parallelism. When processes are delayed while trying to establish communication, they are effectively dead and no further execution can take place until after the send/receive transaction has been completed.

A further issue arises in the specification of the length of time a blocked process must remain inactive. In the initial blocking send proposal, the sending process may unblock as soon as the receiver has accepted the message. In later development however, it has become necessary to provide a reply primitive, where the sender remains

blocked until it receives an acknowledgement from the receiver. Since the sender is still blocked at the time this acknowledgement is sent, such a reply primitive would itself be non-blocking.

In an attempt to alleviate some of the problems associated with synchronous communication and encourage concurrent execution, various forms of asynchronous communication have been suggested.

Buffered message passing allows the sending process to continue as soon as it has deposited the message in a buffer. Processes synchronize only when a process attempts to send a message and the buffer is full, or when a process wishes to receive a message and none is available.

Several complexities are associated with this proposal. In the first place, the size of the buffer is important. With an unbounded buffer, there is no concern for overflow, however the sender can then proceed unhindered and will eventually "outrun" the receiver. On the other hand, in curtailing the execution of the sender by using a bounded buffer, the possibility of deadlock cannot be ignored. This is especially true if the bound is hidden beneath the abstraction.

Secondly, the messages received via a buffer may not accurately reflect the state of the sender [Sch82]. It is impossible to tell how long a message has been stored in the buffer and how many more recent messages have subsequently been sent by the sending process.

An alternative form of asynchronous communication is the conditional primitive proposed in [Gen81]. Using a conditional send or receive, the transaction is completed only if the partner process is blocked waiting. In all other events, the communication fails, thereby ensuring that only messages reflecting the most recent state of the sender are ever transmitted. This scheme is particularly effective in cases where processes can establish communication with a number of different partners. Polling is then required to establish synchronization.

2.5.2. Addressing schemes

Processes taking part in communication need some form of identification so that they can be uniquely specified in the message-passing statements. Two possible means of achieving this have been considered.

Firstly, the process name itself can be used. This implies that process names must be static and where an array of processes is used, an upper limit on the number of processes must be specified at compile-time. Explicit naming falls short in an environment where processes can be created dynamically.

An alternative to direct naming allows for a unique process identification to be allocated at the time a process is created. All subsequent message-passing statements refer either directly or indirectly to this identification. Indirect reference implies that links (or channels) are established between processes and communication proceeds via an available channel number. Using this naming convention, communication between dynamic processes can be supported.

Identifying the processes taking part in communication implies far more than merely determining a naming convention. The relationship between sender and receiver is of particular importance.

In the simplest case, a one-to-one mapping of processes, each sender may send to only one explicitly named destination, which in turn may receive messages from no other source.

A more general mapping is commonly referred to as the client/server model. This is a many-to-one relationship in that one server may receive requests from several client processes. Also known as "receive-any", this addressing scheme has two advantages [Gen81].

1) Requests from several clients can be processed in a non-deterministic order. This encourages parallelism by preventing those processes ready to communicate from being blocked unnecessarily.

2) The use of subroutines from libraries is permitted, since a process using the receive-any semantics need not know the identification of its partner process. Generalized "service processes" lend themselves naturally to inclusion as library routines.

A complication arises though, when the non-blocking reply primitive is used in conjunction with the client/server relationship. Having received a message from an unknown client, the server must send a reply. This kind of asymmetry raises the "return address" problem which can be overcome to a certain degree, by allowing the client to transmit its identification code as part of the message.

The "receive-any" design scheme gives rise to a further problem concerning access control. Particular care must be taken to ensure that all client processes are serviced in such a way as to avoid deadlock and to maximise throughput. Independent communication between client processes must be considered when designing the server routine and thus in fixing the order of transmission [Fra85].

A many-to-many relationship can be achieved using the broadcast send, where the receiver is not explicitly named. Communication is completed either by allowing a pre-determined set of processes to receive the message, or by synchronizing with the first process to issue a receive-any request.

It has been found that there is little use for this primitive as it can be modelled by the other addressing schemes in conjunction with a non-deterministic selection mechanism [Gen81].

A further extension to the available addressing schemes allows for multiple prioritised message queues (or channels) to be associated with each process. Messages are received on a particular channel

depending on their priority, and the process is able to deal with messages from the higher priority queues first.

This extension is rather theoretical and could hardly be used in a real distributed system, where links must be explicitly hard-wired. It would be impractical to allow for more than one physical channel between any two nodes in the distributed system.

2.5.3. Message formats

Where messages are used both for synchronization purposes and to exchange data, processes must be able to recognize and deal with various types of messages. The form of a message should allow for efficiency in establishing and completing the communication and for ease in understanding the semantics of the program.

Two syntactic message formats have been used [Gen81]. In the first instance communication can take place via "function calls", with the message corresponding to the actual parameters of the call. The receiver must recognise the types of the message components by consulting the formal parameter list in the function header.

An alternative arrangement would be to pass the message components as fields in a variant record, such as the one provided by Pascal. This allows more flexibility in that the format of the reply need not be (and in most cases is not) the same as that of the original message sent. The basic variant record must however cater for all possible message types. Consequently extraneous fields containing useless data may have to be transmitted in both directions of the transfer.

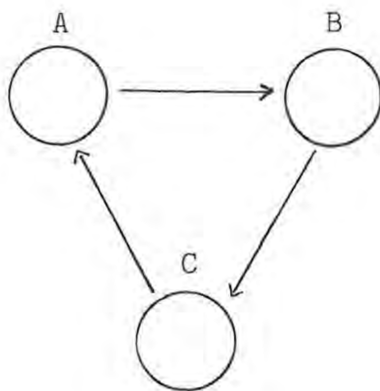
A further consideration in determining the message format is whether the length of the message should be fixed or variable. The former allows for easier implementation in that storage requirements for messages are always constant. Variable length messages however, overcome the problems which can arise when a message, which is larger than the fixed length allocated, needs to be transmitted.

2.5.4. Communication failure

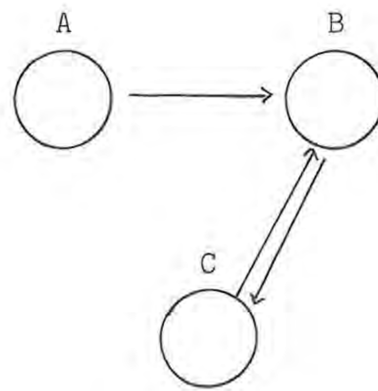
Designing fault tolerant programs is non-trivial. Deadlock should be avoided and checks must be incorporated to take appropriate action if a partner process needed in a communication sequence has previously terminated.

Detecting exposure to deadlock in systems where FIFO channels are used has been shown to be very difficult [Rau85]. When direct naming is used however, processes can be depicted as nodes in a directed graph, with the communication possibilities shown as arcs connecting the nodes. The presence of any cycles leads to a deadlock situation.

Suppose process A wishes to establish communication with process B, which in turn has an outstanding transaction with C. Deadlock arises if process C wishes to communicate with A and not B. We can illustrate this as follows:



Deadlock!



Deadlock free as long as B wishes to communicate with A at a later stage.

The use of bounded graphs to detect deadlock is attributed to Cheriton [Gen81].

Depending on circumstances, an active process attempting to communicate with a terminated process can either be terminated itself or be allowed to restart the sequence of events which led to the failed attempt at communication. In repeating certain instructions, the process may be able to request the services required from another active process.

Care should also be taken to ensure that the same communication protocol is used throughout a program. Consider the chaos arising when one programmer (A) uses the non-blocking reply primitive, and another programmer (B) the blocking-send primitive in the development of their respective processes. Should a process written by A attempt to transmit a message to one of B's processes, it would be blocked forever waiting for the acknowledgement. Timeouts can be incorporated into the implementation of any message passing scheme to prevent deadlock as a result of errors such as these.

Concurrent Languages

3. CONCURRENT LANGUAGES

Numerous message passing schemes can be devised making use of different combinations of the design alternatives discussed in the previous chapter. Recently several language models have been developed which are based on implicit or explicit forms of message exchange. These include CSP, Distributed Processes, Ada, occam, SR, Communication Ports, PLITS, CHILL, Input Tools [VdB81], Port Language [Ker86] and Smalltalk [Gol83].

Most of these language proposals have been implemented; either in their entirety or a modified subset thereof. In the discussion which follows, only those mechanisms for process communication, synchronization and nondeterminism, featured in several of these proposals, will be considered.

3.1. Communicating Sequential Processes

CSP is a notation developed by Hoare [Hoa78, Hoa85] for expressing concurrent programs mathematically. The parallel components of a CSP program are referred to as processes and the sole primitives provided for both synchronization and communication are the input and output commands.

<destination> ! <tag> (<value list>)

causes a message to be sent to the process defined by <destination>, while

<source> ? <tag> (<variable list>)

results in a message being received from process <source>.

Direct naming in the form of static process names, is used to specify the <source> and <destination> processes.

Either simple or composite messages may be exchanged. In both cases the length of the message is variable. <tag> above represents a constructor for a composite message, with the variable and value lists containing the actual message received or transmitted. Single values may be transmitted by an output command of the form

<destination> ! <value>

where <value> represents a simple expression. A corresponding input command is used to complete this data transfer.

In order to establish communication between any two processes, A and B, three conditions must be satisfied.

- Process A (the source) must issue an output command naming B as the destination process.

- Process B must simultaneously declare its intention to input from process A by executing an input command.

- The format of the message output by A must agree in tag (if it is a composite message) and/or both in type and in length with the format of the message which B is waiting to receive.

CSP uses an unbuffered synchronous message-passing scheme, so that whichever process reaches its communication statement first, must wait until the other named process reaches a corresponding statement. As soon as the transfer of data has been completed successfully, both processes are free to continue independently and in parallel.

Nondeterminism in CSP is based on the use of the guarded command which takes the form

<guard> -> <command>

where a <guard> may be composed of a number of Boolean expressions optionally followed by an input command.

The Boolean expressions implement local nondeterminism, since their truth value may be determined independently of any consultation with the other processes. This is in compliance with the traditional definition of nondeterminism introduced by Dijkstra [Dij75].

Global nondeterminism is created by using input guards, since these guards may be resolved only by inspecting the state of the other processes and their willingness to establish communication. [Fra79]

There is no direct way of implementing a client/server relationship in CSP, since both processes involved in communication must explicitly name the other. This restriction can be overcome using an alternative command, which consists of a number of guarded commands from which an arbitrary "enabled and ready" guard is selected for execution.

(A guard whose Boolean expressions are true is said to be enabled, and becomes ready if the process named as the destination in the input guard has expressed its willingness to communicate.)

A server process could thus be programmed as one alternative statement with requests from each of the clients being dealt with by the input command in a separate guarded command. In essence this simulates the receive-any construct without lifting the restriction of the explicit naming of both partners.

3.2. Distributed Processes

Designed by Brinch Hansen [Bri78] at about the same time as Hoare's CSP, Distributed Processes (hereafter referred to as DP) is a concurrent programming concept using remote procedure calls and guarded regions for process communication and synchronization respectively. It was designed specifically to construct distributed processes executing on microcomputer networks without shared memory.

Processes in DP consist of variable declarations (global to the process in which they are declared), common procedures and an initial statement. Communication between concurrent processes is achieved by making calls on these common procedures.

Let process X call procedure DIV in process Y. The command

```
call Y.DIV(in1, in2, out1, out2)
```

would be issued by process X, while the procedure definition

```
proc DIV (x,y : integer #quot, rem : integer)
  begin
   ..... body of DIV ......
  end
```

would appear as a common procedure in process Y.

Transfer of data takes place via parameters in the procedure call. In the above example, data from X to Y is transmitted by the variables <u>in1</u> and <u>in2</u>, while output data from process Y can be accessed in X by reference to the variables <u>out1</u> and <u>out2</u>.

In a system with common storage, this data exchange can merely be done by copying the parameters from one memory location to another. However, in distributed systems for which DP was intended, the message exchange is done by explicit input and output of the relevant parameters.

Once X has issued the call to procedure DIV it must wait until Y accepts the external request and decides to execute the procedure and indeed until after Y has completed the execution of DIV. This means that X remains dormant during the execution of the called procedure thus inhibiting parallelism rather substantially.

As can be seen from the above example, the calling process must

identify the callee, however the reverse is not true. Common procedures need not know the identification of the process initiating the external request. This policy does have its advantages [Gen81], but in some cases the identification of the caller is needed by the common procedure. In the event of this happening, the identification must be explicitly passed as a parameter and the user is in essence duplicating information, which the implementation could ascertain itself.

Nondeterminism is controlled by guarded commands (also based on Dijkstra's guarded command [Dij75]) and guarded regions. The former construct is very similar to its counterpart in CSP, with minor differences in the syntax. A guarded region however, causes any delay in the execution of a process to be dependent on the value of a Boolean condition.

Execution of a DP program begins by concurrently executing the initial statements in all processes. Within an individual process, execution continues thus, until the initial statement terminates or is delayed on some condition. When either of these happen, the process can accept external requests to its shared procedures. Once the execution of a procedure has been started, it in turn continues to completion or until it is delayed. Execution then returns to where it left off in the initial statement or, if this has already terminated, waits for another external request.

This switching between execution of the process body and remote calls to the shared procedures causes even more nondeterminism in DP programs. On completion of an external request, the process body and other remote calls have equal priority in resuming execution. There is the possibility therefore, that execution of the initial statement may be held up indefinitely while remote procedure calls are dealt with [Wel80].

3.3. Ada

Ada, designed for the American Department of Defense [Geh84a, Geh84b], was intended for real-time process control systems. Consequently, it provides high level constructs for multiprocessing and device control.

Influenced by both DP and CSP, Ada provides four basic units from which programs may be constructed. The task is the basic unit of concurrency. Process communication and synchronization are viewed as inseparable activities and are provided by the so-called rendezvous, merely an extension of the remote procedure call of DP.

A rendezvous is initiated by an entry call, which automatically suspends the calling task. The communication is completed when the called task reaches a corresponding accept clause and completes execution of the statements specified therein. This entry/accept mechanism is equivalent to the CSP input/output commands with the added advantage that two-way communication can appear within the rendezvous.

An entry declaration appears in the task specification in much the same syntactical form as a procedure declaration

```
ENTRY div ( <formal parameters> )
```

The transfer of data takes place by passing the actual message as a parameter to the entry call. Tasks in Ada may be nested and this permits another method whereby data may be exchanged. Variables declared in an outer task are binding on all tasks nested within this enclosing task. Within a task hierarchy therefore, information can be exchanged by reading from and writing to these shared variables.

As in DP, the naming convention used is asymmetric, since the calling task names the callee in the entry call, but the accept clause in the body of the entry need not (and in fact does not) know the identity of the task calling it. This encourages the use of "service-processes"

residing in public libraries; but in some instances, it forces the programmer to generate this information using the restricted private type feature.

Nondeterminism is provided by the selective wait statement, which has an optional else clause. Execution of this else-part can cause busy waiting if no feasible accept clause is available [Wel81]. The selective wait statement is modelled on the use of input guards in CSP.

Accept clauses only may appear as guards to the selective wait, thus introducing more asymmetry to the Ada rendezvous. In the original definition of CSP [Hoa78], similar asymmetry is caused by the absence of output guards.

In view of the fact that both the call and accept statements are blocking, mechanisms have been included in the language to control the length of time for which tasks remain synchronized. The conditional entry call is non-blocking in so far as communication only takes place if a rendezvous is immediately available; otherwise the call fails. Blocking on accept can be avoided or minimised by using the COUNT attribute of an entry [Geh84b]. This enables the server to ascertain the number of calls waiting on the entry, and to issue the entry-call only when this number is small.

All Ada tasks are enclosed in the specification of another program unit and automatically begin executing when the body of the enclosing program unit is entered. A task terminates when it reaches the end of its body or if an explicit terminate statement is executed.

3.4. Occam[1]

Occam [May83, Jon85] was designed by Inmos[1] primarily for use on a system consisting of many interconnected micro-computers. The transputer is currently under development to provide the basic building block for such systems [Inm84].

The influence of CSP in the occam programming language is clearly visible and it has been referred to as the official implementation of the CSP notation. The occam syntax tends to be more simple and regular than that of CSP. Indentation is used extensively and primitive commands appear on separate lines. This satisfies one of the design considerations that occam programs be composed with the aid of a syntax checking editor.

Communication between two processes is achieved using input and output commands very similar to those found in CSP; only the naming conventions differ. In occam, processes are identified using channel names, which must be explicitly declared. A channel is a unidirectional connection between two concurrent processes, where one process may input from it and the other may output to it. Two channels are therefore necessary between any two processes to allow two-way communication.

Communication is established when both an input process and an output process are ready to communicate on the same channel. The processes synchronize while the message is being transmitted and then both processes continue executing in parallel. If a process is waiting to input from a number of channels, communication takes place on the first channel used for output by another process.

Besides an equivalent one-to-one relationship between communicating processes, further resemblance to CSP can be found in the use of the

1. Inmos™ and occam™ are trademarks of the INMOS Group of Companies.

guarded command to implement selection. Occam's alternative constructor waits until a message is available on any one of the channels specified in the input guards of the component processes. The first component process found to be ready is chosen and the guard, followed by the guarded process, is then executed.

The IF constructor is similar to the ALT constructor with the exception that Boolean conditions are used in place of input guards. The conditions are tested sequentially and the process corresponding to the first true Boolean is executed. The IF statement terminates when all the Boolean guards are false.

3.5. Synchronizing Resources

Designed by Andrews [And81, And82], Synchronizing Resources (SR) generalizes the mechanisms used by CSP, DP and Ada for interprocess communication. It appears to have a less limited application domain than any of its predecessors since it was designed with the goal of bridging the gap between languages suited to conventional shared memory systems and those intended for use on distributed architectures.

Three new primitives, namely resources, operations and input statements are defined to qualify the relationship between parallel components, communication and synchronization respectively.

A program consists of a set of resources, which resemble monitors in that they define shared variables, one or more processes, and a set of permitted operations. Processes in different resources communicate by means of operations, while those processes in the same resource may interact either through operations or using shared variables.

In order to cater for various forms of process interaction, operations incorporate features of both buffered and unbuffered message passing, and procedure calls. In a way, the communication scheme adopted by SR

is very similar to Ada's remote procedure call, but rather less restrictive.

Operations may have formal parameters, some of which can be used to return results. Declared in the same way as procedures, these operations are activated by either a call or send statement. When using a call statement, which in effect implements synchronous unbuffered message passing, the calling process is delayed until the operation is completed and the results of the operation (if any) have been received.

Invoking operations with a send rather than a call, implies asynchronous buffered message passing. The send statement is non-blocking, thus permitting the invoking process to continue its execution as soon as the parameters have been transmitted. This primitive is used when the calling process merely passes on a value, as is the case in a pipelined computation.

Operations are defined within input statements, which permit selective execution and thus nondeterminism. Boolean expressions, combined with the operation name to provide synchronization, enable these input statements to control which of several available operations may be executed. With the use of arithmetic expressions for scheduling, input statements can also control the order in which multiple invocations of the same operation are executed.

3.6. Communication Ports

The Communication Ports (CP) proposal of Mao and Yeh [Mao80] was designed for programming in a distributed system. It is very similar to DP and Ada in that a procedure call mechanism is used, but two additional capabilities have been provided. The ability to specify exactly which processes may communicate with one another introduces a similar degree of data abstraction to that found in most sequential high level languages. Maximum process overlapping is supported by

enabling the programmer to specify when these processes should be disconnected.

The notion of a "port" is used for communication and synchronization [cf. procedure in DP, entry in Ada]. The calling process is known as the servant, and it attempts to establish communication with the master of the port by executing a connect statement. Communication is set up when the master reaches a corresponding port statement. (The master may be forced to wait at the beginning of a port if no slave is currently suspended on a connect statement.)

The calling process may proceed only after an explicit disconnect statement has been executed by the process with which it is connected. The master process thus has full control over the length of synchronization.

Information is transmitted through the formal parameters declared in the port statement and through the parameters attached to the disconnect statement. In addition, the process identification of the servant is always passed to the master as an implicit system parameter.

A nondeterministic communication port is included in the CP notation to express the possibility of more than one port being activated at a particular time. A condition associated with the port declaration corresponds to the Boolean guards in Dijkstra's guarded command. If a servant process attempts to activate a port, and the condition is false, the port is immediately closed and no communication can take place.

3.7. PLITS

The design of PLITS [Fel79], an acronym for Programming Language in the Sky, was guided by the chief objective of providing a high degree of parallelism in concurrent programs. In the attempt to satisfy this

aim, data sharing in PLITS has been reduced to the absolute minimum and the program modules communicate solely by means of messages.

A further goal in the design of this notation, is the provision of programming techniques to build reliable systems. This is largely achieved by the use of modules, as these are entities with full control over the messages they accept. It is therefore possible to program each module in such a way that it never reaches an undesirable state.

Messages are composed of name-value pairs, called slots, where the "name" part is common to all modules. Message passing is asynchronous and, even where execution of a receive statement may cause blocking, additional primitives have been provided to avoid delays when no messages are available.

When a set of pending messages are available, arrival ordering is used in conjunction with the message attributes to select the next message to be received [Bar83].

Whether the naming convention is to be symmetric or asymmetric is determined by the programmer. Direct communication can be achieved by having both the sending and receiving modules name each other, while the effect of port naming is realized when the receiving module does not name the source. As in CSP though, where a message requires a reply, it must be explicitly constructed and sent by the receiver as a new communication.

3.8. Some General Observations of Message Based Languages

Both advantages and disadvantages can be formulated for each of the proposals discussed in the preceding sections. Some of these may be attributed to the design objectives, while others stem from implementation specifications.

For instance, Ada has been found to exhibit less nondeterminism than both DP and CSP. Moreover, the rendezvous concept utilized by Ada provides a more convenient method of bidirectional message exchange than the separate input and output constructs of CSP.

However, in its use of shared data, Ada has enforced additional communication overheads, which would never occur in an implementation of Hoare's CSP, where processes communicate only by means of explicit input/output commands [Geh82].

The use and expressive power of these languages, in reality merely different combinations of the same constructs, still needs to be investigated. Only by using the languages to develop real systems can their true worth be established.

Although most of the basic problems allied to message passing have been identified and some measure of progress has been made in finding solutions, much still has to be done in respect of communication failure. In addition, the design of protocols for the construction and composition of concurrent processes has only recently been attempted [Bar83]. This concept remains to be investigated thoroughly.

An Implementation of CSP

4. AN IMPLEMENTATION OF CSP

4.1. Original Definition of the Notation

When Hoare designed the mathematical notation CSP [Hoa78], he did so with the view to providing an outline upon which formal specifications of concurrent languages could be modelled. At the time he emphasized that his definition was by no means complete and that further development was necessary.

In the spirit of Hoare's attitude, various extensions and clarifications to the original CSP notation have been proposed.

Kieburtz and Silberschatz [Kie79] broach the subject of termination of processes and suggest an alternative convention for co-operation, whereby the communication and synchronization aspects are independent of one another.

This may be achieved if all communication is routed through I/O ports with sufficient memory to hold a single message. Incorporating this convention, means that the requirement of delaying the sending process until the receiver is ready to accept the message may be relaxed, as long as the sender does not output another message to the same port before the buffer has been emptied.

Advantages of this scheme could be a better assurance that processes will terminate cleanly, as the state of the port may be explicitly tested, and a smaller overhead for the communication facility, since "hidden" signals need never be used.

Bernstein [Ber80] highlights two features of the language, namely nondeterminism and guarded commands, and the problems arising from them. It seems that the association of a priority function with the constituent guarded commands in an alternative command would be a desirable feature to enable the programmer to specify the selection

order in certain algorithms. This function should be able to be modified in response to changes in the state of the process. Nondeterminism would then be allowed amongst guards with equal priorities.

A solution to removing the asymmetry in guarded commands, by allowing output guards, is also given in [Ber80]. The suggested implementation defines a new "query" state, which all processes enter on reaching an alternative command. Depending on the state of the partner process and the availability of an executable guard, the querying process either returns to the active state and executes a guard, or is forced to wait. The reader is referred to Chapter 7 of this thesis, where the algorithm to implement output guards will be discussed further.

In an attempt to provide a more general input/output command, Silberschatz [Sil81] elaborates on the suggestion in [Kie79] that communication should be handled through ports. Ownership of these ports is defined and an abstract implementation of such a port-directed communication scheme is presented. In addition, possibilities for further research, such as port restrictions to be applied in a real implementation and the use of arrays of ports, are briefly discussed.

4.2. Early Implementations

In addition to the need for further study and development of the proposed CSP notation, Hoare recognized the need for an efficient implementation. Besides the implementation of occam by Inmos, the author is aware of only two further attempts in which the original CSP notation has been the major influence. (Numerous other implementations of languages incorporating the features of message passing, but not those explicitly defined by CSP, do of course exist. Mention has already been made of these in a previous chapter.)

In the implementation by Roper and Barter [Rop81] a fairly CSP-like

syntax is used. The semantics attached to the various constructs however, are significantly different from those expressed by the original notation. Automatic buffering is allowed between processes resulting in an asynchronous communication scheme. With regard to the termination problem, processes waiting to input from already terminated processes are themselves not terminated immediately, but forced to remain inactive until all other processes have either completed execution, or are in a similar inactive state.

Messages are composed of a varying number of slots and are manipulated as record-structured objects. In the reception of a message, the identity of the sending process plays no part; only the type of message and the order of its arrival are important. This means that more than one output command may be associated with any input command.

Roper and Barter's implementation consists of a recursive descent compiler, an interpreter and a scheduler, which supervises a round robin scheduling algorithm. Global data structures, notably the process table, are used to depict interprocess relationships at compile-time and also in setting up the process descriptors at runtime.

A further component of this implementation is a message handler which supervises the storage of messages and a suitable queueing mechanism for those messages sent, but not yet received.

A subset of CSP, known as CSP-S, has been implemented by Patnaik and Badrinath [Pat84], and allows for the description of distributed algorithms. This subset includes the use of a guarded command, communication by message passing, and a means for parallel execution of processes. Nested parallelism has been eliminated for the sake of simplicity. As far as the syntax is concerned, it differs widely from that proposed by Hoare and the author feels it lacks the "mathematical look".

The CSP-S compiler uses the recursive descent parsing method to

produce intermediate code in the form of quadruples. An interpreter and a scheduler complete the implementation and together they produce target code resembling simple Fortran statements. Code for each process is produced as a separate subroutine.

Three execution states, namely ready, wait and terminated, are defined by Patnaik and Badrinath in an attempt to simulate the execution of several processes, each running on a separate processor. Once a process has been activated, it cannot be deactivated, until its state changes from ready to either wait (when the process is forced to wait for input or output), or terminated (in which case the process cannot be reactivated).

Several global data structures are used in the implementation of CSP-S and consequently some shared memory must be available in the architecture supporting this system.

4.3. The New CSP-i Grammar

In defining a set of production rules (and therefore the syntax) for a new grammar which would include most of the features of the CSP notation, care was taken to deviate as little as possible from the notation proposed by Hoare in [Hoa78]. Alternate forms were substituted only where constructs were left undefined by Hoare, or where conflicts arose from using a parsing scheme which makes use of only a single lookahead.

Hoare omitted to specify the syntax for declarations of any sort. The syntax adopted in his example programs however, follows the conventions of Pascal [Wir71], and consequently the author has used this as a basis for providing a full set of declarations for CSP-i. In an attempt to limit the number of reserved words used in the grammar, the form of array declarations differs marginally from that of Pascal.

Global constants may be defined using an equivalent of the Pascal const declaration, while message types and signal variables are defined in message and signal declarations respectively. Besides these, no other declarations may occur at the global level.

Local definitions are permitted within a process and also as the first statement in a guarded command. The scope of these variables extends throughout the process or to the end of the guarded command respectively.

A CSP-i program is composed of an optional const statement, optional message and signal declarations and a command construct. The latter consists of one or more parallel processes, and is known as a parallel command. Processes may be nested, however recursion is not supported.

The basic CSP commands, namely assignment, input and output, the null command, the alternative command and an iterative command have been implemented. These all conform syntactically to the CSP syntax given by Hoare in [Hoa78].

Arrays of processes may be declared by adding a range delimiter as a postfix to the process name. In a similar way, a series of identical guarded commands may be represented more concisely by a single command, prefixed by a label denoting the applicable range of the command.

```
*[ (i : 1 .. 5) proc (i) ? x -> <statements> ]
```

allows the five elements of the process array <u>proc</u> to be polled continuously in an attempt to find a process ready to input data.

A large variety of expressions are supported, with the syntax following that used in the C Programming Language [Krn77]. The author adopted the C notation as it supports brevity of expression and also enhances the mathematical impression that Hoare intended for his notation.

Comments may be used by enclosing them within a backslash.

\ This is a comment \

The reader is referred to Appendix A where the full syntax of the CSP-i grammar is given in the form of tagged production rules.

4.4. Deviations from CSP in the Syntax of CSP-i

A few important deviations from the original notation were necessitated by lookahead conflicts caused by the numerous occurrences of parentheses, (), and the use of identifiers as the first symbol in several statements as well as in declarations.

Consider the first source of conflicts, parentheses. Hoare uses these

- in the declaration of arrays, process labels and guard labels,
- in the use of subscripts and instance labels,
- to delimit the components of messages, and
- to construct compound expressions with the correct precedence.

In CSP-i, square brackets have been used to replace parentheses in the declaration of arrays and the subsequent use of subscripts. This enables the parser to distinguish between array elements and members of process arrays.

Curly brackets, {}, are substituted in the declaration and use of all messages and signals.

As a solution to the second source of conflicts, the process name in a process declaration must be preceded by an underscore.

```
[ _process1 ::

      \ body of process \  ]
```

An alternative to using the underscore would be to start a process declaration with a reserved word, say proc. This has been avoided in the interests of preserving the idea of a mathematical notation.

In anonymous process declarations, where no process name is specified, the double colon (::) must be present to demarcate the start of the process body. This is not required in Hoare's specification. The following two declarations, in CSP-i and CSP respectively, serve to highlight this modification.

```
        CSP-i                              CSP

    [ :: a, b: integer;            [ a, b: integer;

      \ process commands  \ ]        \ process commands \ ]
```

The CSP-i Compiler

5. THE CSP-i COMPILER

5.1. Choice of Parsing Method

The CSP-i compiler, written in the C Programming Language, parses by the well known method of bottom-up parsing, and generates a set of intermediate codes during the translation of the language constructs.

The choice of parsing method was based on the fact that an LR-parser can always be produced for a context-free grammar. The same is not true for a recursive-descent parser.

(A context-free grammar can be specified by a finite set of productions of the form

left hand side (l.h.s) -> right hand side (r.h.s)

where l.h.s is a single non-terminal and r.h.s may consist of zero or more non-terminal and/or terminal symbols [Bac79].)

In order to justify the choice of LR parsing over LL parsing, a definition of the CSP-i grammar as a set of production rules conforming to the rules for a context-free grammar, was required. This proved to be a trivial exercise, and it seemed an obvious choice to adopt the more powerful of the two parsing methods.

A further consideration in choosing the parsing method was motivated by the ease with which the parser might be changed if it was constructed with the aid of a parser-generator. Any extensions and alterations to the original CSP-i language could be reflected with the minimum of change to the parser. In fact, for most minor alterations only the tables used by the parser need be regenerated.

In implementing a language proposal such as CSP, it is inevitable that the grammar would have to evolve by trial and error. This merely

reinforced the choice of using an LR parser.

The LR(1) parser-generator, QPARSER[1], was used to generate the tables and control mechanism required by the CSP-i parser. The single lookahead was found to be adequate in parsing the majority of CSP constructs. In the exceptional cases identified in the previous chapter, the CSP-i grammar was modified to resolve unavoidable conflicts.

The LR parsing method and parser generators in general, will not be discussed further in this thesis as these are fully documented elsewhere [Aho74, Brr79]. The reader is also referred to the QPARSER manual [Qca84] as a comprehensive guide to the techniques used by a compiler generated in this way, in parsing and attaching semantic meaning to the various sentences of a grammar.

It should perhaps be mentioned that two alternative methods are provided by QPARSER for manipulating the nodes on the semantic stack, namely the "pointer" method versus the "copy" method. The former has been adopted in the CSP-i compiler, with the result that nodes on the stack are referenced via pointers to dynamic nodes containing semantic data. The "copy" method requires that this data be copied onto the stack and is distinctly less efficient for larger grammars.

The choice of the C programming language has simplified the coding aspect quite considerably. Pointers have been used freely, leading to more efficient manipulation of the stacks and symbol table.

For the most part, the CSP-i compiler is fairly conventional. Those aspects relating to the constructs specific to CSP are likely to be unfamiliar to the reader. In the subsequent discussion of the implementation, reference will be made to the actual source code. Words in uppercase denote explicit identifiers used in the coding of

1. QPARSER is a trademark of QCAD Systems, Inc.

the algorithms and are given in order to facilitate the reader's attempts at following the line of computation in Appendices D and E.

5.2. Declarations and the Use of the Symbol Table

5.2.1. Simple variables

When scalar variables, such as integers, real numbers and characters, or arrays of these types are declared, a new node is created for each identifier. This node is then inserted into the symbol table in a location specified by a hashing algorithm. The type of each variable is also represented in the symbol table. For the sake of efficiency, global pointers, such as INTPTR and REALPTR, have been initialised to refer to a unique entry for each of the standard scalar types. New type nodes are constructed to represent the subrange and base type of arrays.

Symbol table entries for both the identifier and its type are assigned a compile-time level representing the depth of nesting of the block in which they are declared. This compile-time level indicator is incremented whenever a command is parsed in which local declarations may appear. On entering an alternative command for example, the block nesting becomes one level deeper and any variables declared in the guarded commands are distinguishable from all other variables declared thus far. This also simplifies the removal of local declarations on exit from the guarded command.

CSP-i caters for the textual nesting of processes (parallel commands), resulting in the possibility that shared variables may exist. A "disjointness" property however ensures that there is no write access to these variables [Bar83].

5.2.2. Processes

The numerous symbol table entries for process declarations are linked

using three pointers, PARPROC, NXTPROC and LABPTR.

- The first of these, PARPROC, connects a process entry to that of the process which spawned it.

- Following the NXTPROC link, the entries for all declarations of processes with the same name (but naturally with disjoint range labels) may be accessed. The declaration of two processes with the same name and with overlapping ranges will result in a compilation error.

- Finally, LABPTR links the variable nodes representing the identifiers used in the range declarations, if present. Where a single label, instead of a subrange, follows a process declaration, (and consequently where no identifier has been explicitly defined to represent this label) a dummy symbol table entry is created. The symbol name <proclabel> is assigned to this node. This dummy entry is not linked into the hash chains and can only be accessed via the LABPTR of the particular process.

It may appear to be inefficient to include an entry for an anonymous label variable, however this ensures that all labelled processes can be treated uniformly. Compare the following two process declarations:

```
_X (i : 1 .. 3, j : 1 .. 2) ::
```

and

```
_X (Ø, j : 1 .. 2) ::
```

In the first example two label variables, i and j are defined and storage for these is allocated in the stack frame of process X.

The second declaration shows the explicit definition of only one label variable, j. To allow the runtime activation of all instances of process X to be preceded by loading both label values into the

respective stack frames, storage for two variables must be allocated in the latter example as well.

All process entries are assigned the global compile-time level. This prevents their inadvertent removal from the symbol table at any stage. A hole-in-scope rule applies to nested process declarations to ensure that a process name is known in all of the enclosing processes, but not in the named process, or in any of its ancestors. This prevents communication within a process hierarchy and thus bypasses a potential cause of deadlock.

To clarify the use of the various pointers associated with a process node, a graphical representation of process X (as defined in this section) is given in figure 5.1. To assist in the comprehension of this diagram, the structure declarations for the identifier, process and subrange entries in the symbol table are presented below.

```
struct {  /* symt = VARI (identifiers) */
        int saddr;        /* base address */
        boolean canchange;            /* distinguishes label vars */
        struct symtabtype *vtypep,    /* its type */
                          *nxtvar;    /* used by label+mess vars */
       } vsymt;
struct {  /* symt = scalar types and SUBRANGE */
        struct symtabtype *subtype;    /* its type */
        int size, udim, ldim;
       } asymt;
struct {  /* symt = PROC_TYPE (processes) */
        int procnum,      /* number of first instance of process */
            noprocs,      /* total no. of instances of process */
            locals,       /* no. of local vars */
            dim;          /* dimension of label */
        boolean defined;
        struct symtabtype
               *labptr,  /* identifiers in label */
               *nxtproc, /* next process with same name */
               *parproc; /* previous process */
       } psymt;
```

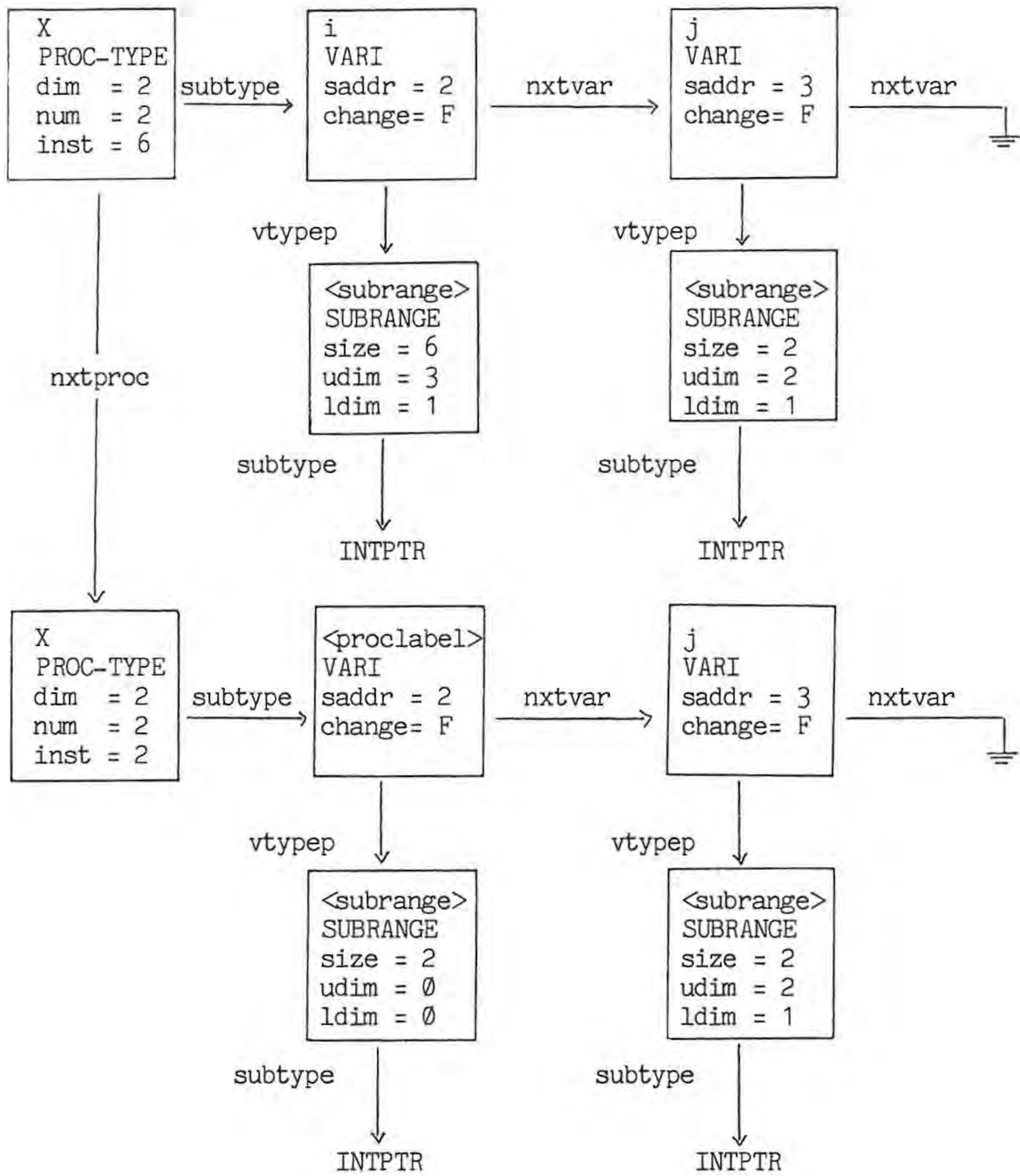


Figure 5.1

5.2.3. Message declarations

All messages constructed with an explicit constructor must be globally declared under the message section of a program. The syntax of this section is

```
message
    <cons1> { <components> } ,
    <consn> { <components> } ;
```

where n > 0, <$cons_i$> is the constructor associated with the i'th message, and <components> is a list of zero or more component types. Arrays of messages are not permitted. Neither may any component of a message be a message itself; that is, message declarations may not be nested.

In the following declarations,

```
message
  pickup { integer, integer },  putdown { integer, real };
```

pickup and putdown are in essence "type" declarations and can be used to declare variables or "instances" of these message types:

```
a, b : pickup;
c, d : putdown;
```

Although such instance declarations improve the readability of CSP-i programs, they are not compulsory, as the formal message constructors may be used directly in input/output commands and in message assignments. Local variables, representing the message components, must be declared if this latter option is used. This will be clarified towards the end of this section.

An entry in the symbol table for a message declaration contains the following fields:

```
struct {  /* MESS_TYPE (messages) */
          struct symtabtype *subtype;
          int size, udim, ldim;
        } asymt;
```

Associated with each message constructor is a unique identification number, contained in the UDIM field of a MESS-TYPE node in the symbol table. This identifier is used in input and output commands where the message formats of both sending and receiving processes must agree. Where no constructor is used in a communication command, the messages are matched according to the number and types of their components.

The LDIM field in the symbol table entry for a message contains the number of message components, while SIZE represents the total stack space required by a message of this kind.

The constructor node of a message is bound to the nodes of its components using the SUBTYPE link of a MESS-TYPE node. A "dummy" name <messitem> is used to represent each of these components since only their types are specified in the declaration. The <messitem> nodes are in turn linked to one another using the NXTVAR link in a VARI node. The SADDR field of these dummy variables is used to specify the offset of the actual component within the storage allocated to the message as a whole.

Consider the above with reference to the following example.

```
message
        a {real, integer};
```

At this point the entries added to the symbol table can be depicted by figure 5.2.

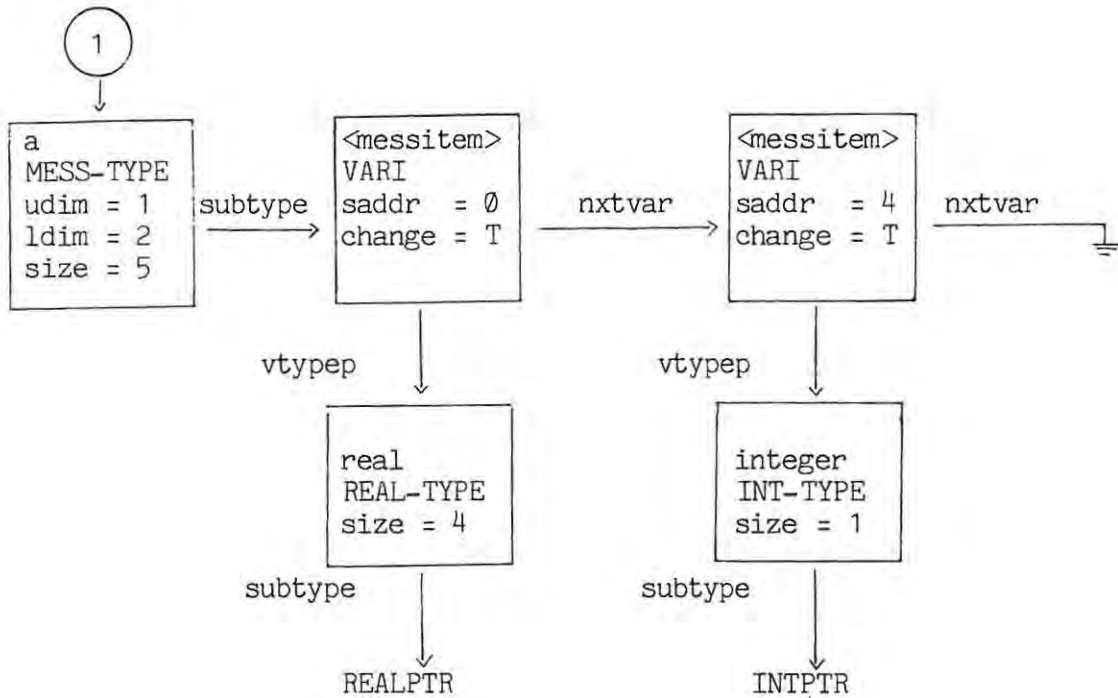


Figure 5.2

In the event that instances of this message are declared, say

```
p1, p2 : a;
```

the symbol table entries for these (given in figure 5.3) would be appended to the entries shown in figure 5.2 above.

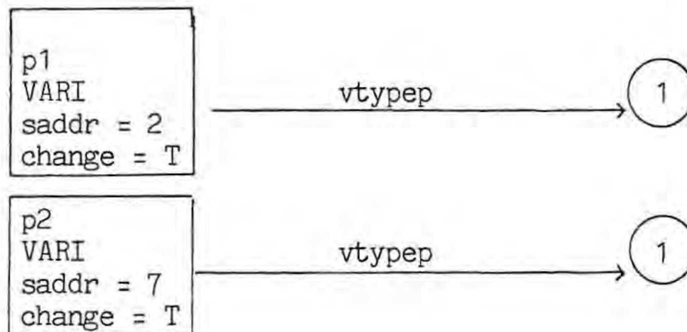


Figure 5.3

The first real component of p1 can be found at offset 2 in the local stack frame of the process in which it was declared, while the integer

part of the message is located at offset 6. The latter offset is calculated as

```
  2  (being the relative start address of p1)
+ 4  (the offset of the actual component).
```

Where message type variables are used directly, that is, without having declared any instances, it is necessary to declare local variables corresponding to the number of components of the message type. Using the message declaration for a (as given above), two local variables must be declared local to any process using this message.

```
i : integer;
r : real;
```

The storage allocated to these two local variables (which need not be contiguous) represents the actual storage for a when it is used with components r and i. The SADDR field of the <messitem> node is undefined in this case.

5.2.4. Signals and their declaration

Signals are messages without components, and are used solely for synchronization purposes. No data transfer takes place when signals are transmitted.

Signals need not be explicitly declared. Any constructor used in a process body and which is undeclared at the time, is entered into the symbol table at the global level and given a unique message number. All subsequent uses of this signal variable will be referenced through its id-number.

The author felt that the explicit declaration of signals was time-consuming and unnecessary and hence the provision of a default method of declaration. Since signals are merely a simplified form of message, they may be declared in the message section of a program,

thus ensuring uniformity in the syntax of the language. Identifiers declared in this way are type variables and can be used either by declaring instances of them or by referencing the constructor with an empty component list. This is in keeping with the declaration of message types described in section 5.2.3.

A third alternative is provided for the declaration of signals, that is as global variables of type signal.

```
signal   end, begin;
```

This method of declaration facilitates the use of signals and simplifies the understanding of CSP-i programs, since a single variable name end can be substituted for the phrase end{}. With the inclusion of the global signal declaration given above, the following two input commands would be identical.

```
kbd ? end;

kbd ? stop{};  \ where stop has not previously been declared \
```

The difference between the message and signal sections of a CSP-i program should now be apparent. Identifiers declared in the former section are similar to the Pascal type declarations, while those in the latter section are simply global variables.

5.3. Parsing Selected CSP-i Constructs

5.3.1. Assignment of structured types

Arrays and strings may be directly assigned to array variables, provided that the target array variable is large enough to contain the source variable or string. Two signals may be assigned to each other even if their constructors differ. This is an ineffective assignment since no data is actually transferred and consequently no intermediate

codes are generated.

Assignment of one message to another may only take place after the following conditions have been satisfied:

- both messages have the same constructor, which may be empty,
- the size of the message is the same for both operands, and
- the types of the individual components of the messages agree.

Considering the declaration of message types and the possible declaration of variables of these types, four distinct syntactical forms of message assignment are possible. Using the declaration

```
message cons {real, integer};
```

these can be identified as

1. the mutual assignment of two message variables

```
x, y : cons;
x := y;
```

2. the assignment of one message to another, without using message variables

```
a,c : real;
b,d : integer;
cons{a,b} := cons{c,d};
```

3 and 4. the assignment of a message variable to the individual components of a message type and vice versa

```
a : real;
b : integer;
x : cons;

cons {a,b} := x;
x := cons {a,b};
```

In the remainder of this section, the semantics of these assignments and the way in which the compiler generates intermediate codes for each of them, will be considered.

The transfer of data from one message variable to another constitutes the simplest case, because both variables must have been declared locally and therefore contiguous storage has been allocated to their components. In much the same way as an array assignment, the complete message is merely copied from one storage location to another.

Generating code for assignments corresponding to the second type identified above, is also fairly trivial. A message assignment of this kind can be decomposed into one or more simple assignments. In this example, these are

```
a := c;        \ and \
b := d;
```

where a, b, c and d have each been allocated storage individually by virtue of their local status. No storage is allocated to the message as a whole.

In the third case, the message variable x is locally declared and at the same time sufficient storage to contain all the components is allocated. Although two assignments are made, they are not as precisely defined as in the previous case.

Using the offset of variable x (referred to as adrx) and the offsets of the components of the message type cons (adr1 and adr2 for the first and second components respectively), the individual assignments are

```
a := contents of (adrx+adr1);
b := contents of (adrx+adr2);
```

or, in the reverse assignment

```
contents of (adrx+adr1) := a;
contents of (adrx+adr2) := b;
```

By now it should be clear to the reader that an assignment between messages provides a concise method of doing multiple simple assignments. Consequently only one fundamental code generation algorithm is necessary to cater for all assignment statements.

The basic code generated for a simple assignment is

```
load the address of the target variable on Top of Stack (TOS)
load the value of the expression on TOS
store the expression on TOS at the address given by (TOS-1)
```

In the compilation of message assignments, instances of this basic code will be generated according to the number of message components.

5.3.2. Input and output commands

The input and output commands in CSP-i are the sole means of effecting communication and synchronization between parallel processes. When an input command corresponds to an output command, data transfer takes place.

The following two commands in processes A and B respectively,

```
B ? {x, y};
```

and

```
A ! {3*c, 8+d};
```

have the combined effect of assigning

```
x := 3 * c; and y := 8 + d;
```

The algorithm used in generating code for communication commands is

given below:

load the expressions to be output or the addresses of the
 target variables on TOS
calculate the identification of the partner process and load
 this value on TOS
emit a send or receive opcode respectively
generate the message format

Evaluating the target expressions in an output command, and the addresses of the target variables in an input command, proves to be relatively simple and straightforward. It should be noted that in the transfer of messages, the last component of the message is evaluated first. This ensures that data exchange proceeds in the order specified textually by the program.

The message format is generated as an array of specifications denoting the types and sizes of the individual components and is emitted as a series of integers. Where a constructor is used to compose a message, the unique identification number associated with this constructor is included in the message format. This ensures that precise matches are made when choosing from a number of available messages for transmission.

The generation of the partner process number is non-trivial. In the early stages of this project, a unique number was allocated to each individual process and members of process arrays at the time of their declaration. This scheme is adequate provided that redefinition of processes with disjoint ranges is not permitted. It is easy to see where the method falls short if the restriction on redefinition is not enforced.

Consider a program which implements a pipelined communication system:

```
[ _X (i : 1 .. 3)  ::
      a : integer;

      X (i-1) ? a;                    \ ------> 1 \
      X (i+1) ! a

//
  _X (0) ::

      X (1) ! 100                     \ ------> 2 \

//
  _X (4) ::
      a : integer;

      X (3) ? a;                       \ ------> 3 \
      screen ! a
]
```

If we were to assign identification numbers to processes at the time of compilation and in the order that they have been declared, X(1) would become process 1, X(2) process 2, X(3) process 3, X(0) process 4 and X(4) process 5. This is clearly undesirable.

In generating the process-id at position 1 above, the value of 1 would be used as the starting number for the declared instances of process X and then at runtime, the i component would be added to this. Unless the process declarations were done in ascending order according to their range labels, this numbering algorithm would fail miserably.

The method currently used by the CSP-i compiler allows the computation of process number to be done at runtime. All processes with the same name are given a generic process-id and the instance labels are passed to the interpreter along with this identification.

In translating a process name in a communication command, the compiler thus generates code to .

evaluate and load the n'th label expression on TOS
evaluate and place the first label expression on TOS

identify and emit the generic process identification
emit the number of labels in the process declaration

It has already been mentioned that the CSP-i implementation allows a process to communicate with all other processes except those within which it is nested. Following the PARPROC links attached to process nodes, all the ancestors of a process may be traced and communication with any of these excluded.

CSP-i also supports communication with standard devices.

```
kbd ? a;
```

allows a value to be read from the standard input device, while

```
screen ! 100;
```

enables a value to be printed to the standard output device. This extension facilitates user intervention in CSP-i programs.

5.3.3. The guarded command

Dijkstra's guarded commands [Dij75] (incorporating both Boolean and input guards) are used in CSP-i to introduce selective execution and to control nondeterminism. The syntax used deviates only marginally from that proposed by Dijkstra.

The execution of a guard proceeds from left to right. Local declarations may appear as the first entry, and their scope extends to the end of the current guarded command. Thereafter, one or more Boolean expressions are evaluated and if all are found to be true, an optional input command is attempted. Execution of the guard fails if at any stage a Boolean is false or if the partner process in the communication has terminated. In cases where the partner process has not terminated, but is not in a position to establish communication immediately, the guard becomes "pending" and will be retested at a

later stage.

Guarded commands may have ranges associated with them, so that

```
[ (i:1 .. 4) Gi -> Si]
```

is equivalent to

```
[ G1 -> S1 [] G2 -> S2 [] G3 -> S3 [] G4 -> S4 ]
```

In generating the intermediate codes for a single guarded command an algorithm along the lines of the following is used:

```
generate a label at the start of the first guard
evaluate the first Boolean condition
emit a jump-on-false to the fail address
evaluate the n'th Boolean condition
emit a jump-on-false to the fail address
code to execute an optional input guard
code to execute the "then" part of the guarded command
jump to the address associated with successful execution
```

(The reader will notice the absence of a jump-on-false opcode after the code for the input guard has been generated. This is justified since the flow of control after the execution of any communication command is handled automatically by the scheduler.)

Numeric labels are used by the compiler to mark certain locations as the destinations of jumps. As the intermediate codes are written to a file immediately after being generated, no backpatching can be done by the compiler. The LOC intermediate code, with a label number as its operand, is thus used to enable the preprocessing phase of the interpreter to fill in the actual addresses of jump destinations.

In translating individual guards, the label associated with the start of the guarded command is important in that a jump table must

eventually be constructed to contain an entry identifying the start of each guarded command parsed.

In the case where a guarded command has a range, the actual code for the body of the command is produced once only; n entries for this guarded command will appear in the jump table (where n = upper limit - lower limit of range + 1). The code generation algorithm used to translate a multiple guarded command is the same as that given earlier, with the exception that an initial store must be done before any of the Boolean expressions are evaluated. This ensures that the appropriate value is stored in the subrange variable, and all subsequent references to this variable can be treated in the same way as accesses to other local variables.

Jump table entries are constructed as follows:

AJP <destination> n x_1 .... x_n

where the AJP code specifies an entry in the jump table, <destination> is the label associated with the start of the particular guarded command, n is the number of label dimensions and the x_i are the subrange values applicable to this guard.

Consider the guarded command

```
[   (i: Ø .. 1, j : 2 .. 3)
        a > 5; proc(i,j) ? a  ->  C1
 []
      a <= 5  ->  C2     ]
```

A jump table representing these commands would contain the following entries

```
AJP label1 2 Ø 2
AJP label1 2 Ø 3
AJP label1 2 1 2
AJP label1 2 1 3
AJP label2 Ø
```

5.3.4. Alternative and repetitive commands

An alternative command consists of a number of guarded commands, only one of which is executed. If all guards are false after an initial test, the alternative command fails and the process aborts.

Repetition in selection is introduced using the repetitive command, which continues executing the enclosed alternative statement until the latter fails. If all guards in the alternative command are initially false, execution of the repetitive command is equivalent to a skip command.

In the discussion on guarded commands, reference was made to a fail address and the address associated with successful execution of an alternative command. These can now be defined with reference to the code generation algorithm for an alternative command.

```
loc1    code to mark the start of the alternative statement
        <code for guarded commands as given earlier>
loc2    label at the beginning of the jump table
        <contents of jump table>
loc3    label to mark the end of the jump table
        code to abort the process
loc4    label to mark successful execution
loc5    <code for the next command>
```

Loc3 is demarcated as the fail address, so that if one guarded command fails, another can be chosen from the entries in the jump table. If

all guards have been tested and all are found to be false, execution falls through to the unconditional jump, thus exiting the alternative command.

The reader will notice that the success address and the address of the first command after the alternative command are given by two separate labels; yet in the above algorithm, they represent the same code address. Both are necessary when the alternative command is enclosed in a repetitive command, in which case additional code is generated to control the looping aspect.

The algorithm then becomes

loc1	code to mark the start of the alternative statement
	<code for guarded commands as given earlier>
loc2	label at the beginning of the jump table
	<contents of jump table>
loc3	label to mark the end of the jump table
	unconditional jump to loc5
loc4	label to mark successful execution
	unconditional jump to loc1
loc5	<code for the next command>

From this it should be clear that when the alternative command fails, the jump to loc5 ensures that the repeat loop is also terminated.

5.4. Suspension of Code Generation

The variable, GENCODE, is used in the compiler to inhibit the generation of intermediate codes by allowing code to be produced only when the variable has a zero value. Code generation is suspended as a result of syntax errors in a program, in which case GENCODE becomes negative 1, or when the compiler parses a guarded command which will always fail. This can only happen if one of the Boolean guards can be

evaluated at compile-time and it produces a false result.

When such a situation arises, the value of GENCODE is incremented only if it was previously zero, thus preventing a double increment if two guards are found to be permanently false. At the end of the guarded command, GENCODE is decremented once if it has a value greater than zero.

Nested guarded commands pose somewhat of a problem. If GENCODE has been incremented in the outer guarded command due to a false guard, it must automatically be incremented at the start of the nested command. This ensures that when the value is decremented at the end of the inner command it will still reflect the falsity of the outer guarded command by having a positive value. No intermediate code will therefore be produced for the nested alternative command or for the remainder of the outer guarded command.

5.5. An Alternative Implementation for CSP-i

After the completion of the CSP-i compiler using the LR(1) parsing method and the C Programming Language for coding purposes, the author devised a possible alternative scheme for implementing the translation and interpretation mechanism of a CSP-orientated language.

This scheme was inspired by the similarity of the structure of the program fragments in [Hoa85] to that of LISP programs [Win84]. Taking advantage of such a similarity, CSP constructs could be directly translated into equivalent LISP functions and executed using an existing LISP interpreter.

There is a drawback to this theory concerning the execution of processes in parallel. Current LISP systems do not support concurrency; however, a language known as MULTILISP has been designed to provide the facilities for implementing parallel execution of symbolic constructs. At present this language is being implemented on the 32-processor Concert multiprocessor [Hal85].

With the availability of a concurrent implementation of a LISP-like language, the alternative scheme proposed above would be feasible and perhaps more efficient that the current CSP-i implementation.

In retrospect, it seems that even without a concurrent LISP system, the translation of the CSP constructs into an intermediate language resembling LISP might have simplified the project. Further research needs to be done however, to ascertain the validity of this.

The CSP-i Interpreter

6. THE CSP-i INTERPRETER

Once a CSP-i program has been compiled and found to be free of syntax errors, the interpreter may be invoked. Together the compiler and interpreter effect the execution of concurrent programs on a single IBM PC.

The CSP-i implementation does not support distributed algorithms or a multiprocessor architecture. Further extensions would be necessary to incorporate distributed computing, and these will be considered in the final chapter.

The interpreter may be subdivided into three interdependent phases, namely preprocessing, scheduling and intermediate language code handling.

6.1. The Preprocessor

All intermediate language codes (ILC's) are written to a file immediately after being generated, with the result that no backpatching of jump destinations can be attempted at compile-time. The preprocessing phase enables the interpreter to read the intermediate language codes from a file into a code array, filling in forward references and incomplete definitions at the same time.

A further use of the preprocessor is in the creation of a dynamic ring of process descriptors, which contains one node for each defined process or process instance, in the case of an array of processes.

The descriptor of process i contains the following information:

- execution status of the process (STATUS)
- links to the descriptor node of process i+1 (NEXT) and to the node of the process which spawned i (PARENT)

- communication information, comprising a link to the descriptor of the partner process (MATE), as well as a message format (FORMAT) and the direction of the data transfer (DIREC)
- pointers to the dynamic ring of guard descriptors, used while interpreting alternative and repetitive commands (GUARDRING and CURGUARD)
- the number of processes spawned by i (SIBS)
- various identification fields including a unique number (UNID) allocated at runtime, and a family number (NUM) together with process labels (LABS), allocated by the compiler
- program counter (PPC), stack pointers (PPB, PPT and STACKEND), address of the first code instruction for process i (FIRSTINST) and a display of all stack bases associated with the runtime levels (DISPLAY).

The reader is referred to Appendix E (file CSPIDEFS.C) for the full declaration of the process descriptor (DESCRIPT). Additional fields not mentioned here are introduced by the implementation of output guards and these will be discussed elsewhere.

The astute reader may be concerned by the potential "duplication" of identification information in that a unique number as well as a process number with instance labels is allocated to each node. In the previous chapter, justification was provided for the scheme of emitting process names in the form of a process number with a label. Any process name used as an operand in a ILC can only be identified in the process ring using a similar labelling mechanism. The unique process number however, not only serves as a simplified means of identifying descriptors, but is also vital in the implementation of output guards (cf chapter 7).

It is important to note that at the time a descriptor node is created for a process, no stack frame is allocated and the process is not yet capable of being executed.

6.2. The Scheduling Procedure

In view of the fact that concurrent programs are ultimately executed on a single processor, the most one can expect in terms of parallel execution is a form of quasi-parallelism created by time-slicing. The scheduler thus simulates pseudo concurrency by allocating these time-slices to the processes on a round-robin basis, taking into account the state of the process and its ability to execute at the time. Time slices are quantified as a small random number of ILC's to be interpreted.

Associated with each process is an execution state which can be either READY, SUSPENDED, or TERMINATED. On creation a process is suspended and only becomes ready to run once the parallel level at which it has been compiled is reached. A process remains in the ready state until completion (in which case the status changes to terminated) or until it is forced to wait on input or output. This latter event causes the process to become suspended and also initiates an immediate process switch.

Three further states namely QUERY, IDLE and WAIT are necessitated by the inclusion of output guards. Interaction with the standard input device while interpreting a guarded command calls for an additional state, the USELESS state. These will be dealt with in the following chapter.

The process descriptor ring forms the very heart of the scheduler, since it contains all the information needed for process switching and message transfers.

In the event of no ready processes being available for execution, but when not all have terminated, deadlock is detected and the program aborts with an appropriate message.

To simplify the detection of deadlock, a dummy process descriptor exists in the process ring describing the "main" process. There is no

lexical equivalent of this process, however all "real" processes are effectively spawned by it. In identifying the success of a CSP program, attention need only be paid to the status of the descriptor node for this dummy process.

6.3. The Intermediate Language Code Interpreter

Each ILC consists of a mnemonic followed by a variable number of numeric operands. These opcodes are based to a large extent on those given in [Qca84]. A complete list of the CSP-i intermediate language codes, together with their format and a brief description of each, is given in Appendix B.

After the successful completion of the preprocessing phase, all ILC's (bar 3) may be present in the code array constructed by the preprocessor. (LOC, DEF and LAB are considered preprocessor directives and are used solely in setting up the process descriptor ring and in backpatching jump destinations.)

Using the information contained in the process descriptor ring, the interpreter is able to execute each of the ILC's generated by the compiler.

6.3.1. Basic structure of the interpreter

The basic routines in the CSP-i interpreter are based on those in the Clang implementation [Cha84]. Written in the C Programming Language, the code of the interpreter requires 56K and runs adequately with a heap of 64K. Naturally the dynamic storage requirements increase as the number of concurrent processes rises.

The interpreter uses only one memory stack for all local variables as well as storage for temporary results. The entire array allocated as the runtime stack is carved up into equal stack frames and one of these frames is allocated to each process at the time it is activated.

No alternative storage is provided in the event of overflow within a stack frame or where an attempt is made to allocate more stack frames than are available. In the event of either of these happening, the program will abort with a termination message.

All mathematical and logical ILC's are evaluated by loading either one (in the case of unary operators) or two operands onto the top of the local stack frame and then applying the appropriate operator. The stack is then collapsed to reveal the result. Thus the simple addition, a + 4 (where a has the value 5), would yield the following stack manipulations:

Opcode	Stack
	<empty>
LDI (offset of a)	5
PSI 4	5 4
ADI	9

Operations on real numbers pose no problems if the stack is incremented and decremented according to the size of the type of variables involved. Furthermore the generalized use of pointers in C allows for an efficient manipulation of real numbers on the stack, as well as in the code array.

Checks for subscript bounds which are out of range, division by zero and non-integral modulus are included and result in runtime execution errors.

A variety of load and store operators are provided. These include loading integers or real numbers from an address at the Top of Stack (TOS) (LOI, LOR), or from the address given by the operand (LDI, LDR); storing values at the address at the TOS (SOI, SOR), or at an address given by the operand (STI, STR); loading strings onto the stack (LDS); loading or storing arrays either at an address given by the operand (LDA, STA) or an address on the TOS (LOA, SOA) and simply loading an

address (LID). The operands for these ILC's are described in Appendix B.

6.3.2. Activation and termination of processes

As has already been described, every process instance has a node in the descriptor ring containing information relevant to its execution. Most of the information can be initialized at the time the node is created. The data in a few fields however, only becomes available after the process has been activated.

Before the intepreter actually begins evaluating ILC's, all processes declared at the global (or outermost) level are allocated a stack frame and their status is set to READY. At the same time the dummy global process will be suspended and its number of sibling processes set according to the number of outer processes activated.

The scheduler is then in a position to choose a ready process as a starting point in the execution of the program.

All nested parallel processes at a particular level and with the same parent process are activated simultaneously as soon as a BEG mnemonic for any one of them is interpreted. As described above, a stack frame is then allocated for each of these processes, enabling the outstanding descriptor fields in the process nodes to be completed. The parent process is suspended and one of the newly activated siblings is allowed to start executing.

Whenever a process terminates naturally, the sibling count of its parent is decremented. Once this value reaches zero, the parent is re-activated and allowed to resume execution. At no time will a process node actually be removed from the process ring as there is a possibility that is may be called again on say, a second pass through the loop in which it was declared. A mere change in the status of the process is enough to reflect its demise.

A CSP-i program terminates successfully if all its constituent processes terminate. Once all the processes at the outermost level have terminated, the sibling count of the dummy global process is reduced to zero, and this process is re-activated. The only opcode in the body of the main process, HLT, is then executed, causing the dummy process to terminate and hence the CSP-i program as well. Andrews uses a similar centralized termination algorithm in the implementation of SR [And82].

A novel feature which allows for the design of concise concurrent programs, but which is absent in several other implementations of concurrent languages, is the availability throughout the process ring of the knowledge that a particular process has terminated. (The absence of this knowledge is particularly noticeable in the implementation of occam by Inmos [Inm84b].)

In allowing such knowledge to be available, a situation referred to by Roper and Barter in [Rop81] as "quiescence" can be avoided. This phenomenon arises when all non-terminated processes are suspended on communication with terminated processes. In a quiescent situation, the system devised by Roper and Barter allows all processes to be terminated as they maintain "that no useful computation can occur".

It is the author's opinion that this termination scheme is not generally applicable. In most cases where an input or output command (which is not part of a guard) appears in a repetitive command, there is an alternative flow of control which, if executed, would not involve the communication command which may originally have caused the quiescent situation.

Consider the following program fragment in process Z.

```
*[   a >= 10   ->  X ! a; kbd ? a
   []
     a < 10    ->  Y ! a; kbd ? a   ]
```

Suppose process Z is suspended on the output command, X ! a, and process X terminates. According to [Rop81], no further computations would be useful. However, until process Y terminates, any value less than ten could be passed on to Y and computations within the process could proceed normally. The fact that X may have terminated prematurely would have no effect on the successful execution of Y, if this process was, for example, computing the average of numbers less than ten.

"Useful" calculations as illustrated by this example are not provided for in the implementation described in [Rop81].

In the CSP-i implementation, knowledge of the termination of X is immediately passed on to Z, enabling the latter process to fail any input or output commands involving process X. Even if Z is suspended on such a communication with X at the time of the termination of the latter process, execution in process Z is allowed to recommence at the next command.

6.3.3. Interpreting input and output commands

Two forms of communication may be executed by the interpreter, namely communication with standard devices and message transfer between processes. The former allows user intervention in the execution of a program. All interpreter and scheduler activity is suspended by the operating system for the duration of the data transfer when a user process interacts with one of these standard devices and the communication command is not a constituent of a guarded command. Interaction with the standard input device while interpreting a guarded command does not cause the entire system to delay. This point will be discussed in detail in chapter 7.

Communication between user-defined processes does not result in program suspension except in the case of deadlock, where the program is aborted by the scheduler. Only the process initiating the communication need ever be suspended and only until such a time as a

suitable partner can be found to complete the rendezvous.

The interpreter adheres to the three golden criteria defined in [Hoa78] for selecting a suitable partner in a message communication.

The first requirement states that each process must explicitly name the other in the communication command. This can be realized simply and effectively by maintaining a link from the descriptor node of the suspended process (that is, the first of the pair of processes to execute a communication command) to the node of the requested partner process. As soon as the latter process executes either an input or output command, the correspondence between the communication requirements of the two respective processes may be tested.

To ensure that the communication commands specify opposite directions for the transfer, a direction field is associated with the partner link. This data identifies the role of the suspended process in the message transfer. A simple comparison of two Boolean fields serves to satisfy the second criterion.

Finally, the formats of the messages required by the communicating processes must be identical. This entails a trivial comparison between the operands of the input/output mnemonics of the two processes. (The reader is reminded that the communication mnemonic is followed by a series of integers reflecting the type, size and constructor (if present) of the message.)

Once a suitable combination of processes has been established, the message transfer takes place. Since the CSP-i implementation involves a shared memory architecture only, this transfer can be done by merely copying the appropriate values from the stack of the source process into the locations specified by addresses on the stack of the destination process. No acknowledgements are sent to either process upon completion of the transfer; the synchronization between the two processes is terminated by the interpreter and thereafter both are free to proceed individually and in parallel.

It has already been mentioned that a communication statement fails if one of the partner processes has terminated. In this event the stack frame of the active process is collapsed to eliminate all temporary values resulting from the translation of the unmatched communication command.

6.3.4. The alternative and repetitive commands

Given the following iterative selection

```
*[  (i : 1 .. 3)  proc(i) ? x -> C1
   []
    proc(0) ? x -> C2                ]
```

the intermediate code generated by the compiler would be something along the lines of

```
LOC LAB1                   ;location 1
ALN LAB6

LOC LAB4                   ;location 4
store label variable
code for input guard
code for C1
UJP LAB5

LOC LAB7                   ;location 7
code for input guard
code for C2
UJP LAB5

LOC  LAB6                  ;address of jump table - location 6
JBL LAB3
AJP LAB4 1 1
AJP LAB4 1 2
AJP LAB4 1 3
AJP LAB7 0
LOC LAB3                   ;fail address - location 3
EBL
UJP LAB2

LOC LAB5                   ;success - location 5
UJP LAB1                   ;repeat loop
LOC LAB2                   ;command fails - location 2
```

The execution of an alternative or repetitive command can be illustrated, with reference to this code sequence, by examining the interpretation of each of the major opcodes in turn.

6.3.4.1. The ALN opcode

Execution of an alternative command commences with a transfer, caused by the interpretation of the ALN opcode at location 1, to the start of the jump table, location 6. The flow of control at this point proceeds in any one of two directions; either sequentially through the ILC's or by skipping the AJP codes and interpreting the EBL opcode immediately.

In order to explain this split in the flow of control, it is first necessary to outline the effect of interpreting the AJP opcode.

6.3.4.2. AJP intermediate language codes

During the interpretation of the AJP codes, a dynamic ring with a dummy head is constructed to represent the entries in the jump table. Informally known as the guard ring, this data structure facilitates the execution of a guarded command and controls nondeterminism in an alternative command.

Each node in the guard ring, excluding the head node, contains the following information

- a link within the ring (GNEXT)
- a status associated with the guard (FAILED)
- the dimension of a possible range (DM) and the applicable range labels (GLABS)

The dummy node at the head of the ring does not represent any particular guarded command, but rather contains information about the state of the alternative command as a whole. Additional data fields

are included in the structure of this node to record

- the total number of guard nodes in the ring, and thus the number of guarded commands in the alternative command (NODETOT)
- the number of guards that have been tested and how many have failed (TESTED and FAILTEST respectively)
- the fail address of the alternative command, so that each guard ring can be uniquely identified (FAILADR)
- pointers (PREVHEAD and PREVCUR) to an outer guard ring, that may exist if the current command is nested within another alternative command.

6.3.4.3. JBL opcode

The direction of the flow of control after interpreting ALN is directly determined by the operand to the JBL opcode, which contains the unique fail address of the alternative command in question. If a guard ring already exists and whose identification matches this fail address, the individual nodes need not be reconstructed. All that remains to be done is for the various control fields in the head node and the status field in the individual guard nodes to be reinitialised to their starting values. Initially all guards are untested and none can have failed.

In this way, a guard ring remains intact for repetitive passes through an alternative command.

Alternatively, if no matching guard ring can be found, the flow of control proceeds sequentially through all the AJP codes, constructing the dynamic ring of nodes in the process.

In the event of nested alternative commands, it is clearly advantageous to optimize the creation and destruction of the guard structures. Compare the repetitive command and the ILC's given earlier in this section, with the nested alternative command

```
* [   a > 10  -> [  b > 10   -> C1
                  []
                    b <= 10  -> C2  ]
    []
      a <= 10  ->  C3    ]
```

and the following intermediate codes

```
LOC LAB1
ALN LAB6

LOC LAB4
code for Boolean guard
FJP 3

ALN LAB10                  ;marker 1
LOC LAB8
code for Boolean guard
FJP LAB7
code for C1
UJP LAB9

LOC LAB11
code for Boolean guard
FJP LAB7
code for C2
UJP LAB9

LOC LAB10
JBL LAB7
AJP LAB8 0
AJP LAB11 0
EBL
EXT                        ;marker 2

LOC LAB9
UJP LAB5

LOC LAB12
code for Boolean guard
FJP LAB3
code for C3
UJP LAB5

LOC LAB6                   ;address of jump table
JBL LAB3
AJP LAB4 0
AJP LAB12 0
LOC LAB3                   ;fail address
EBL
UJP LAB2

LOC LAB5                   ;success  - marker 3
UJP LAB1                   ;repeat loop
LOC LAB2                   ;command fails
```

On reaching the second ALN opcode at marker 1, a guard ring already exists for the outer repetitive command. A second ring is then needed to represent the nested alternative command.

Considering that at least one further pass through the outer loop is inevitable, the state of the outer ring should be maintained for the sake of efficiency. By allowing the PREVHEAD and PREVCUR pointers in the head node of the inner ring to record the existence of the outer command, this optimization is effected.

Two further intricasies that arise in the interpretation of the alternative command are highlighted by this code sequence. Firstly, after every Boolean guard there is a jump-on-false to the fail address, to enable the execution of a guarded command to be aborted as soon as one of its constituent guards is found to be false.

A second point that might not be completely obvious on first reading, is the presence of the EXT opcode at marker 2. This opcode enables the process to be aborted if all the guards in a purely alternative command (as opposed to a repetitive command), are initially false. It should perhaps be mentioned that no EXT is generated if the alternative command is enclosed within a loop. The execution of such a repetitive command, with the same initial values for the constituent guards as before, resembles a skip command and does not result in the termination of the entire process.

6.3.4.4. The EBL mnemonic

Irrespective of the path taken after interpreting the JBL opcode, both channels converge at the opcode used to designate the end of the jump table. The interpretation of this opcode is responsible for starting the mechanism of selection and ultimate execution of a guarded command.

Starting at a random node in the guard ring, a linear search is conducted to select a node, the corresponding guard of which has not

yet been tested or which has not yet failed. Having found such a guard, and depending on whether it fails or not, either the selection procedure is repeated by returning to the EBL opcode, or the successful execution of the guarded command allows the execution to proceed beyond the jump table to the next executable CSP-i command.

6.3.5. Interpreting input guards

Where only Boolean guards are used in a guarded command, it is trivial to determine the validity of the guard simply by executing the Boolean expression and then testing the result. Input guards cannot be tested in this way, since the execution of the input command when the partner process is not suspended on a corresponding communication command, can cause the entire process to be delayed. This would result in the first enabled guard always being chosen for execution and another guard, which is both enabled and ready, being bypassed.

In his specification of the CSP notation, Hoare explicitly states that "an implementation should take advantage of its freedom of selection to ensure efficient execution and good response". In particular that "[the input guard] which corresponds to the earliest ready and matching output command should in general be preferred; and certainly, no executable and ready output command should be passed over unreasonable often" [Hoa78].

The method of testing Boolean guards and denoting their status in the guard nodes is evidently not feasible for use with input guards and an alternative method needs to be devised.

The scheme used in the CSP-i implementation classifies an input guard as failed, if, and only if, the partner process in the communication has terminated. If the guard is enabled (through the presence of a true Boolean) but not ready, the status of the guard becomes PENDING. This allows the input command either to be awakened and retested at the time the partner process is ready to communicate or to be ignored if another enabled and ready guard has been selected.

The reader is referred to the discussion of the implementation of output guards for a more detailed insight into the mechanism used for selection within a symmetric alternative command.

6.3.6. Other features of the interpreter

Garbage collection is carried out whenever dynamic lists are destroyed. This occurs primarily in the use of the guard ring, where every execution of a nested alternative command and the first execution of all outer alternative commands requires the construction of a dynamic ring. Use of the backpatch table by the preprocessor also requires storage allocated from the heap and this is similarly reclaimed once the table has served its purpose. No reclamation of the storage used by the process table is necessary as this structure remains intact throughout the execution of the program. Although processes may terminate, their descriptor is never physically removed from the ring as there is the possibility that a process may be reactivated.

Primitive debugging facilities are provided by the two opcodes STK and DMP, which are generated on parsing the '@' and '^' characters respectively. These characters may be placed freely in the CSP-i source code. STK causes a decimal dump of the stack frame of the current process, while DMP gives a listing of all processes defined within a program, together with some relevant data about each process. This data includes the status of the process, stack frame usage, current communication links and the value of the program counter. The generation of a process listing is especially useful in solving deadlocked programs.

Finally, a trace facility, which echoes all ILC's interpreted, as well as details of the process switches, enables the user to monitor the execution of a program.

Refinements
&
Extensions

7. REFINEMENTS AND EXTENSIONS TO CSP

In an earlier chapter, mention was made of some of the proposals for extensions to the original definition of the CSP notation. Justification for these proposals stems from a variety of sources, but of greatest significance is the attempt to provide the programmer with a more general and flexible notation for expressing concurrent algorithms. Of the three extensions incorporated into the CSP-i implementation, the provision of a totally symmetric and nondeterministic selection command offers the most support to this attempt.

7.1. Clean Termination of Processes

It is assumed that all processes in CSP will terminate within a finite time on conclusion of the respective process body. A repetitive command included as a command within the process body however, could lead to restrictions being placed on this initial assumption.

In [Hoa78], a repetitive command is defined to terminate when all Boolean guards have failed and when the partner processes specified in input guards have all terminated. Bearing this in mind, there are two possible ways of ensuring the proper termination of a repetitive command and the avoidance of deadlock in an implementation of CSP.

In the first scheme, the onus rests on the programmer to ensure that provision is made to allow all processes in his CSP program to terminate. Consider the CSP-i version of an example given by Hoare to illustrate how this may be done.

```
[_DIV :: continue : boolean;
      continue := true;

      *[ continue; X ? end{} -> continue := false
       [] x, y : integer;
         continue; X ? {x,y} ->  .....; X ! {quot, rem}
       ]

  //
 _X ::
      \ user interface here \
      DIV ! end{}
]
```

Kieburtz and Silberschatz argue that this restriction is in direct conflict with the definition of the repetitive command and its termination given earlier in this section [Kie79].

The second method affords a process wishing to communicate a glimpse of the status of the partner process. Thus the termination of any process is broadcast to all other active processes and in this way, the implementation itself provides the mechanism to fail corresponding communication guards when a termination message is received.

As was mentioned in the section dealing with the CSP-i interpreter, this latter method has been used in implementing CSP-i. As a result the coding of several concurrent algorithms is simplified since all unnecessary "termination" signals are avoided. An example of a CSP-i program with its occam equivalent is given in Chapter 8 to illustrate this simplification.

In [Hoa85] a completely new approach to the termination problem is presented. Instead of expecting all processes to terminate, Hoare suggests that in view of the implementation complications enforced by this restriction, alternative measures should be sought. It is suggested that non-terminating recursive processes used in conjunction with multiple shared resources could provide a solution.

A similar argument is presented by Milne and Milner who consider the

behaviour of non-terminating processes only [Fra79].

7.2. Implementation of a Generalized I/O Command

7.2.1. Justification for the use of output guards

In the original specification of the CSP notation, the use of output guards was excluded for the sake of simplicity. However, at the same time Hoare presented two arguments in favour of removing this simplification.

Firstly, the expression of certain concurrent algorithms in CSP, notably the "producer-consumer" problem, would be simplified. The CSP-i solution to this problem is given in section 7.2.5. Secondly, a symmetric alternative command would ensure that the behaviour of all parallel constructs could be modelled by sequential commands.

For example, the sequential alternative command which mirrors the parallel command

```
_Z :: [ :: X ! 3 // :: Y ! 4 ]
```

needs to be coded as

```
_Z :: [ X ! 3 -> Y ! 4 [] Y ! 4 -> X ! 3]
```

This cannot be done in an implementation where output guards are not supported. Hoare points out that the following alternative command is not an accurate reflection of the given parallel command.

```
_Z :: [ true -> X ! 3; Y ! 4
      []
        true -> Y ! 4; X ! 3
      ]
```

Deadlock can result if, for example, the first guarded command is chosen for execution, and the processes are synchronized in such a way that process Y must input from Z before process X is allowed to do so.

7.2.2. Alternative schemes proposed

In a system which permits only input guards, the synchronization issues inherent in the communication are relatively simple. Taking advantage of the asymmetry of the I/O command, the rule that every process wishing to input must wait for a corresponding output command, results in a deadlock free implementation. This enables input guards to be handled very efficiently as communication is always initiated by the other process, which cannot also be executing a guarded command at the time. In the initial CSP-i implementation model, this rule was applied to input guards allowing them to be matched only after the partner process had been suspended on a corresponding output command.

If output commands are to be permitted as guards, this simplified implementation scheme fails. The issues involved in synchronizing generalized I/O commands become very complex. In view of the advantages associated with the use of output guards, however, several schemes for a successful implementation have been proposed in the literature.

In [Sil79] a master-slave model of communicating processes is described, where communication may only take place between a subordinate process and its master. This enables both input and output commands to appear as guards, and synchronization is achieved by forcing the process designated as the slave to wait for the master to complete the transaction. Silberschatz does not specify any criteria as to how the master-slave relations should be decided, however suggests that this should be left to the programmer.

A similar model to the one described above is given in [VdS81] where the processes allowed to communicate within an alternative command may be either only the children, or a combination of the parents and

siblings of the enclosing process. An hierarchial tree of processes is used to establish the relationship between processes.

Bernstein's proposal in [Ber80] places no restrictions on the processes named in communication commands; any process may communicate with all other processes. Synchronization is achieved by manipulating communication commands according to the various states a process may have, namely ACTIVE, WAIT, and QUERY.

On entering an alternative command a process assumes the query state. If no executable guard is found the process is forced to wait; otherwise it returns to the active state.

If a process i in its attempts to validate a guard, queries another process j, which is also in the query state, the unique process identification numbers of the processes concerned are used to settle the indecision. Where i > j, process j responds with a busy signal, otherwise it delays responding to the query by process i until it has successfully completed its own query phase.

In Bernstein's proposal, no information is stored regarding the state of any of the processes. This does however lead to some inefficiency, as no upper bound can be placed on the number of probes needed to be sent in order to establish communication between any two processes.

Such is not the case with the solution in [Buc83], where the same priority-ordering algorithm as described by Bernstein is used. Buckley's algorithm makes provision for retaining a limited amount of state information. This means that when process i receives a busy signal from process j, the former process attempts no further communication with the latter until after j has finished its initial queries and has sent a signal to that effect to process i.

Process i is then in a position to remedy the non-committal answer received earlier, by sending a RETRY signal. A definite YES or NO will follow depending on the new state of process j. Only one retry

message need ever be exchanged, and thus the upper bound on the time taken for two processes to establish communication is equal to the time elapsed until the last retry is sent.

Very recently Natarajan described a distributed synchronization scheme for implementing input and output guards. This makes use of dynamic priorities associated with the communication transactions themselves, rather than with the processes. In accordance with the proposals of Silberschatz [Sil81], communication takes place via input and output ports, and consequently this proposal is not entirely suited to the CSP-i model. The reader is referred to the paper by Natarajan [Nat86], for the algorithms used in implementing such a generalized port-directed communication scheme.

7.2.3. Output guards in CSP-i

The implementation of output guards in CSP-i is based on both the proposals in [Ber80] and those in [Buc83]. As the CSP-i implementation has been designed for an architecture with a central processor and shared memory, no explicit signals need be transmitted. The state of a process can be derived by consulting the process descriptor table and the decisions concerning communication are then made by the scheduler.

Before this scheme can be described, it is necessary to clarify the new states and structures used in the implementation of output guards.

Two additional dynamic structures are associated with each process, namely the query and asked lists. The query list of a process i contains an entry for each of the processes which has issued a query to i. Details of the query as well as the environment of the querying process at the time of issuing the signal are stored. The asked list retains the identification of the processes which have been queried by process i.

The new process states which need to be introduced are the following:

- QUERY, which is assigned to a process on entering an alternative command

- IDLE, which a process enters once all guards have been tested but not all have failed

- WAIT, which forces a process switch if a process is delayed on receiving a response to a communication query, and

- USELESS, which is assigned to a process for the duration of the execution of a default guard.

Given below is the algorithm for establishing whether communication may take place between process i, whose communication command is currently being interpreted and which appears within a guard, and process j.

(It should be noted that this algorithm is not valid if the guarded command wishes to establish communication with the standard input device. A modified algorithm for handling this exception is given in section 7.2.4.)

```
if (process j is suspended on a corresponding command) then
   Proceed with the data transfer
   Remove the queries (if any) made by process i from the query
    lists of all other processes        {1}
   Reactivate process j

else if (process j has already queried process i) then        {2}

      if (process j is IDLE) then
        Suspend process i on process j
        Remove the queries to other processes made by process i
```

```
  Process j re-enters QUERY mode and is allowed to immediately
   re-evaluate the guard containing the communication command
   with process i      {3}

else if (process j is USELESS) then
      Suspend process i on process j
      Remove the queries to other processes made by process i
      Process j is allowed to complete the execution of the
       default guard and immediately thereafter must re-
       evaluate the guard containing the communication
       command with process i      {4}

  else if (process j is also in QUERY mode) then

        if (i > j) then
           Process j responds with a non-committal answer
           Allow current guarded command in process to
            be retested at a later stage      {5}
           Process i continues testing remaining guarded
            commands

        else  { i <= j}
           Remove the query by process j to process i
           Reinitialise the relevant guarded command in
            process j to be untested
           Allow process i to query process j      {6}

else  {process i has not yet been queried by j}
   if (process j has terminated) then
      FAIL guarded command in process i
   else
        Set status of guarded command to be PENDING
        Let process i query process j      {7}
```

A few additional comments are given below to clarify the coding of those parts of the algorithm followed by numbered comments.

1. The processes queried by process i can be identified using the asked list of descriptor i. Scanning the corresponding query lists of these processes allows the removal of all references to process i.

2. Ascertaining whether a process has been queried by another can be done by consulting the query list of the former process.

3. When process j queries process i, the program counter of j is retained in the entry in the query list of process i. If a communication match is subsequently found, process j, which initialised the querying, is allowed to resume execution at the guarded command which contains the reference to process i. If a Boolean guard is present in this command, it must be retested, although the truth value will never differ from that found previously. This is ensured as no variables can actually be modified in the condition-part of an alternative command and global variables are not permitted in CSP-i.

The reason why execution cannot start at the input guard itself is attributed to the use of labelled guarded commands. To ensure that the correct label values are stored in the label variables prior to executing any of the constituent guards, execution must commence at the very beginning of the guarded command.

4. The status of process j is not changed in order to allow the execution of the default guard to proceed unhindered. On completion of the otherwise option however, the INRUPT field of the descriptor of process j will contain the address of the guard to be executed next.

5. Enabling a guarded command to be retested at a later stage simply involves decrementing the counting variables associated with the number of guards tested and those that have failed.

6. The three steps in this "else" clause effectively nullify the query by process j and reverse the querying action so that it appears that

process i has initiated the query. This avoids using a busy waiting scheme to delay process i until process j has finished its querying phase.

7. Entries in both the asked list of process i and the query list of process j are made when process i queries j.

This concludes the discussion on how two processes may be synchronized when the communication command of at least one of the processes is also a guard. All that remains to be done is to give the algorithm used when the communication command of process i is not a condition in a guarded command.

```
if (process j is suspended on a corresponding command) then
   Proceed with the data transfer
   Reactivate j
else if (process j has terminated) then
        Fail the current communication command
     else if (process j is in QUERY mode) then
             Process i waits until process j has finished its
              initial queries and then re-executes this
              communication command
          else if (process j has queried i and is now IDLE)
                  Suspend process i on process j
                  Change the state of process j to QUERY
                  Allow process j to retest the guarded command
                   containing a communication guard involving
                   process i
               else if (process j has queried i and is now USELESS)
                       Suspend process i on process j
                       Allow process j to complete execution of the
                        default guard and then to retest the
                        guarded command containing a communication
                        guard involving process i
                  else
                      Suspend process i on process j
```

By comparing the two algorithms given in this section, it should be clear that a communication command, which is also a guard, can never cause the process to be suspended. If this were to happen, it would result in the execution of the initially chosen guarded command to the exclusion of all other ready guards.

Where the I/O command does not form part of a guard however, no progress in the execution of the process is possible until the data transfer takes place. Thus the process may readily be suspended until a partner process is available to complete the exchange.

7.2.4. Guards communicating with standard input

Direct communication with standard I/O devices results in suspension of the host operating system until the required device is ready to exchange data. Such a situation is not at all desirable in the interpretation of guarded communication commands, since it would destroy all fairness currently associated with the implementation of the alternative command. If a guard communicating with a standard input device were to be selected for testing, the input guard would immediately be executed and no further program execution would be possible until the appropriate input data was received. This would mark a successful iteration of the alternative command containing the input guard and no other guarded commands would be tested.

To reintroduce fairness and to allow the first ready and enabled guard to be selected for execution, a "system" process has been created to run in a pseudo-concurrent environment with the other user processes. Designed to act as a process interface between the user defined processes and the standard input mechanism provided by the operating system, this input process is linked into the process descriptor ring and allocated time-slices accordingly.

The role of this process is to poll the standard input device and to record the number of keypresses. At the same time, it monitors the

user processes to ascertain whether any of these have made queries for standard input from within guarded commands. This querying mechanism is done along the lines of the following algorithm. (Assume process i is querying the standard input process.)

```
if (std input process is suspended on a corresponding command)
  then
    Proceed with the data input
    Decrement the number of input signals received thus far
    Remove the queries (if any) made by process i from the query
     lists of all other processes
    Reactivate the standard input process

  else {standard input not ready}
      Set status of guarded command in process i to be PENDING
      Let process i query standard input process
```

Once a query has been received, and provided that the user has entered an input signal by way of a keypress, the system process may answer the queries on a FIFO basis. This is done by either suspending itself and allowing the user process to complete the exchange, or by allowing the data input to take place immediately if the user process is suspended at the time.

The input signals entered by the user are not used as input data per se, but merely serve as an indication that standard input will be available immediately the guard requires it. Thus to allow the guarded command

```
kbd ? num -> C1      \ where num is an integer \
```

to be selected for execution, the user would first need to press a key to declare his willingness to transmit data, and once the input prompt (:) has been displayed, would then enter the integer value.

It should be noted that communication with standard input outside of guarded commands does not involve this system process and is handled directly by the host operating system.

The reader should be aware of the potential complications arising from the use of a single keyboard to provide standard input for a number of user processes. If a number of processes or guarded commands require input simultaneously, there is no guarantee that the data entered at a particular point in the execution of a CSP-i program will be matched to the corresponding communication command.

7.2.5. Simplified CSP-i programs using output guards

It has been mentioned that the "producer-consumer" problem can be simplified by the use of output guards. Given below are two versions of this problem, with and without the use of output guards respectively. The original CSP example is due to Hoare [Hoa78].

\Example 1\
\Producer-Consumer problem without the use of output guards\

```
signal more;

[ _buf ::
    buffer : [Ø ..9] integer; in, out : integer;

     in := Ø; out := Ø;
     *[ in < out + 1Ø; producer ? buffer[in % 1Ø] -> in := in + 1
          []
        out < in; consumer ? more -> consumer ! buffer[out % 1Ø];
                                     in := in + 1
          []
        out < in; consumer ? more -> consumer ! buffer[out % 1Ø];
                                     out := out + 1
      ]

   //
  _producer ::

     *[ true -> produce-item; buf ! item ]

   //
  _consumer ::

     *[ true -> buf ! more; buf ? item; consume-item ]

]
```

\Example 2\
\Producer-Consumer problem incorporating output guards\

```
[ _buf ::
    buffer : [Ø ..9] integer; in, out : integer;

     in := Ø; out := Ø;
     *[ in < out + 1Ø; producer ? buffer[in % 1Ø] -> in := in + 1
          []
        out < in; consumer ! buffer[out % 1Ø] -> out := out + 1
      ]

   //
  _producer ::

     *[ true -> produce-item; buf ! item ]

   //
  _consumer ::

     *[ true -> buf ? item; consume-item ]

]
```

As a further example of how output guards may be used to simplify the coding of an algorithm and to decrease the nondeterminism within an alternative command, consider the pair of processes, of which the first computes the location of a moving point to be displayed on a screen and the second continuously refreshes this display [Kie79].

```
\Example 3\
\Screen update program where the user has no control over the
 selection of guards\

 [ _update ::
     x, y : integer;

   *[ true -> get-coordinates; display ! {x,y} ]

  //

    _display ::
      x, y : integer;

   *[ update ? {x,y} -> skip
     []
      true -> screen ! {x,y}
    ]

  ]
```

```
\Example 4\
\Screen update program using output guards and a buffer process. The
 original CSP version is given in [Ber80]\

 [ _buffer ::
     x, y : integer;
     b : boolean;

      b := false;    \ wait for first update \

    *[ update ? {x,y} -> b := true
      []
       b; display ! {x,y} -> b := false
     ]

   //

   _update ::
     x, y : integer;

   *[ true -> get-coordinates; buffer ! {x,y} ]

   //

   _display ::
     x, y : integer;

   *[ buffer ? {x,y} -> screen ! {x,y} ]

  ]
```

7.3. An Otherwise Option for the Alternative Command

Modelled on the ELSE clause in an Ada SELECT statement, and the proposals by Schneider in [Sch82], an otherwise guard has been added to the CSP-i language.

The otherwise guard can be used effectively in a situation where a process continuously polls several other service processes, but should not be blocked if none of these processes are ready to communicate. Adding an otherwise guard to the alternative command which controls the polling routine allows the process to continue active execution until a service becomes available.

The guard otherwise will not be chosen for execution unless none of the "real" guards in the alternative command are both ready and

enabled. Thus a process executing an alternative command which contains an otherwise option cannot be delayed. Consider the execution of the following command in process Z:

```
Z :: [ X ? i -> C1  [] Y ? i -> C2 ]
```

If neither process X, nor process Y are suspended and awaiting communication at the time these guarded commands are being tested, process Z will become IDLE. By adding an otherwise option to this alternative command,

```
Z :: [ X ? i -> C1  [] Y ? i -> C2
       []
        otherwise -> C3 ]
```

process Z cannot be delayed, as the commands in C3 would be executed, thus terminating the alternative command successfully.

In a repetitive command, the otherwise option will be executed continuously while no other guard is ready and enabled, provided that not all the guarded commands have failed.

The use of an otherwise guard does not change the pattern of execution in a repetitive command if all the guards are initially false. Failure of an alternative command, which includes an otherwise option, will not cause the process to abort. In the event of either of the above, the otherwise clause will not be executed.

In implementing this option, the opcode AJD (instead of the usual AJP) is used to relay information concerning the default guard to the interpreter. Distinguished thus from the "real" guards, no node for an otherwise guard appears in the guard ring. Instead, information about a default guard is retained in the head node of the ring.

Deciding when to execute this option is a trivial exercise. As soon as all guard nodes have been tested, and when not all have failed, the

interpreter transfers control to the "then-part" of the otherwise guard and execution proceeds as for any other guarded command.

A slight complexity arises in the execution of a repetitive command containing both input/output guards and an otherwise guard. In the CSP-i system described thus far, deadlock cannot be prevented if the communication statements corresponding to the I/O guards are themselves guards in a repetitive command containing an otherwise option. Such repetitive commands will remain in the QUERY state for the duration of their existence, thereby preventing any committed communication.

To eliminate this and to ensure a deadlock-free environment, a process assumes the USELESS state while interpreting a default guard. If during this time, a second process responds to any of the communication queries sent earlier, the first process completes its execution of the otherwise option, and immediately thereafter, transfers execution to the input/output guard matching the answered query. The control mechanism used in this case is similar to that applied when the initial process is IDLE at the time of the response.

Where no responses to earlier queries from I/O guards are received during the interpretation of a default guard in a repetitive command, control passes to the top of the loop. All previous communication queries are annulled and a further iteration of the alternative command is started anew.

Critical Evaluation & Conclusion

8. CRITICAL EVALUATION AND CONCLUSION

8.1. The Basic CSP-i System

In the preceding chapters, the author has described a language, based on Hoare's CSP notation, together with its implementation. This marks the successful completion of the project and the fulfillment of all the design goals.

The CSP-i syntax is sufficiently similar to that of the original CSP notation to enable any CSP algorithm to be coded and executed with minimal modifications. It could even be argued that CSP-i programs are easier to comprehend, since all ambiguities resulting from the excessive use of parentheses, have been eliminated.

The CSP-i implementation has effectively proven the soundness of Hoare's notation for use on a single processor architecture. This was originally done using mathematical techniques [Apt80, Lev81].

Of greater practical interest, the CSP-i system has provided a powerful high level language which may be used to illustrate the mechanisms of synchronization, communication by message passing and nondeterministic selection. Most of the well-known concurrent algorithms have been successfully coded in the CSP-i notation, and, as far as can be ascertained, their execution is reasonably efficient. No benchmarks for other implementations of languages based on message passing and executing on the IBM PC are available to the author. The following results however, were obtained for 50 iterations of the parallel matrix multiplication algorithm, where a 3x3 matrix is multiplied by a 1x3 array.

CSP-i on the IBM PC	131 seconds
occam on the Sage IV	59 seconds

As a further design objective, it was the author's intention to implement as many of the suggestions made by Hoare to ensure a fair implementation. The random selection of guards to be tested, as well as the matching of communicating processes on a first-in first-out basis, contribute to achieving this.

In his paper [Hoa78], Hoare touches on a solution to the termination problem, stating that any communication statement, whose partner process has already terminated, should be failed. This philosophy has been followed meticulously in implementing CSP-i and even enlarged upon by allowing terminating processes to broadcast information concerning their demise throughout the process ring. It is the author's opinion that inclusion of this feature has greatly improved the usefulness of CSP-i and from a programmer's point of view, has made it the better choice of a number of message-based languages.

For the discerning reader, it should suffice to compare the CSP-i code for the "Dining Philosophers" problem and the occam equivalent.

```
\ A CSP-i version of the dining philosophers problem, based on the CSP
  solution given by Hoare [Hoa78].  This program terminates when a key
  is pressed.  All processes are allowed to run to completion. \

 [ _ROOM ::
     running : boolean; occup : integer;

    occup := Ø;  running := true;
    *[ (i:1 .. 5) running; occup < 5; PHIL(i) ? enter{} ->
                                           occup := occup + 1
      []
       running; terminate ? end{} -> running := false
      []
        (i: 1 .. 5) running; PHIL(i) ? exit{} -> occup := occup - 1
     ]

   //

   _TERMINATE ::
     finished : boolean;

     finished := false;
    *[ ~finished; kbd ? {} -> finished := true; ROOM ! end{} ]

   //

   _FORK (j: 1 .. 5) ::

    *[ PHIL(j) ? pickup{} -> screen ! {'F',j, 'U', 'P', j};
                             PHIL(j) ? putdown{};
                             screen ! { 'F', j, 'D', 'P', j}
      []
       PHIL(j % 5 + 1) ? pickup{} ->
                            screen ! { 'F', j, 'U', 'P', j%5+1};
                            PHIL(j % 5 + 1) ? putdown{};
                            screen ! { 'F', j, 'D', 'P', j%5+1}
     ]

   //

   _PHIL (k: 1 ..5) ::

    *[ ROOM ! enter{} -> FORK(k) ! pickup{};
                         FORK(((k+3) % 5) + 1) ! pickup{};
                         screen ! {'E', k};    \ philosopher eats \
                         FORK(k) ! putdown{};
                         FORK(((k+3) % 5) + 1) ! putdown{};
                         ROOM ! exit{};
                         screen ! {'T', k}      \ philosopher thinks \
     ]
 ]
```

```
-- The Dining Philosophers in occam, attributed to [Cla86]
-- ----------------------------------------------------

-- This version of the Dining Philosophers terminates when a key is
-- pressed.  All processes are allowed to execute to completion.

chan GETLEFTFORK [5], GETRIGHTFORK [5] :
chan RELEASELEFTFORK [5], RELEASERIGHTFORK [5] :
chan MAYIENTER [5], MAYILEAVE [5] :
chan EAT [5], THINK [5] :
chan DIES [5] :
var FINISHED [5], STOP :

seq
  STOP := false
  seq i = [0 for 5]
    FINISHED [i] := false
  par
    par i = [0 for 5] -- one process for each philosopher
      while not FINISHED [i]
        seq
          THINK [i] ! any
          MAYIENTER [i] ! any
          GETLEFTFORK [i] ! any         -- grab left hand fork
          GETRIGHTFORK [i] ! any        -- grab right hand fork
          EAT [i] ! any
          RELEASELEFTFORK [i] ! any     -- put down left fork
          RELEASERIGHTFORK [i] ! any    -- put down right fork
          MAYILEAVE [i] ! any
          if
            STOP
              seq
                DIES [i] ! any
                FINISHED [i] := true


    par i = [0 for 5] -- a process for each fork
      var NOTFINISHED:
      seq
        NOTFINISHED := true
        while NOTFINISHED -- fork i is philosopher i's right fork
          seq             --            philosopher i+1's left fork
            alt
              GETRIGHTFORK [i] ? any       -- philosopher i uses fork
                RELEASERIGHTFORK [i] ? any -- until he puts it down
              GETLEFTFORK [(i+1)\5] ? any -- philosopher i+1 uses fork
                RELEASELEFTFORK [(i+1)\5] ? any
              skip
                skip
            NOTFINISHED := false
            if i = [0 for 5]
              not FINISHED [i]
                NOTFINISHED := true


    var NUMINROOM, NOTFINISHED :
    seq -- Process to control the number of people in the room
      NUMINROOM := 0
```

```
  NOTFINISHED := true
  while NOTFINISHED
    seq
      alt
        alt i = [Ø for 5]
          NUMINROOM < 4 & MAYIENTER [i] ? any    -- only 4 in room
            NUMINROOM := NUMINROOM + 1
        alt i = [Ø for 5]
          MAYILEAVE [i] ? any
            NUMINROOM := NUMINROOM - 1
        skip
          skip
      NOTFINISHED := false
      if i = [Ø for 5]
        not FINISHED [i]
          NOTFINISHED := true


var NOTFINISHED:
seq
  NOTFINISHED := true
  while NOTFINISHED      -- Process to report on what philosophers
    seq                  -- are doing.
      alt
        alt i = [Ø for 5]
          THINK [i] ? any
            seq
              str.to.screen ("PHILOSOPHER ")
              screen ! i+48  -- Numeric to Ascii conversion
              str.to.screen (" THINKS*C")
        alt i = [Ø for 5]
          EAT [i] ? any
            seq
              str.to.screen ("PHILOSOPHER ")
              screen ! i+48  -- Numeric to Ascii conversion
              str.to.screen (" EATS*C")
        alt i = [Ø for 5]
          DIES [i] ? any
            seq
              str.to.screen ("PHILOSOPHER ")
              screen ! i+48  -- Numeric to Ascii conversion
              str.to.screen (" DIES - SHAME*C")
        skip
          skip
      NOTFINISHED := false
      if i = [Ø for 5]
        not FINISHED [i]
          NOTFINISHED := true


seq                      -- Process to terminate program
  keyboard ? any
  STOP := true
```

By examining the occam solution it can be seen that termination code is needed in every process to ensure that no process is left hanging indefinitely. This is not the case in the CSP-i solution. Enabling one process to terminate cleanly eliminates the possibility of dangling processes, since termination signals are allowed to ripple through the remaining active processes.

In addition, the reader will notice the complexity and verbosity of the occam code, caused by the law of occam programming that only a unique pair of processes may communicate on any one channel. This implies that only a single user process may exchange data on each of the standard input and output channels and all other user processes are prevented from communicating directly with these devices. A "reporter" process is required to act as an interface between the user processes and the screen in this example, and hence the need for the arrays of channels (EAT, THINK and DIES).

In fulfillment of a final design objective, a completely symmetric communication command has been provided for CSP-i. It has already been shown in this thesis how the inclusion of output guards facilitates the coding of several concurrent algorithms, such as the "producer-consumer" problem. An even greater motivating factor for the inclusion of this facility is the wide-spread acceptance (in the literature dealing with the CSP notation), of the generalized input/output command as a very necessary and even primitive feature of the language.

8.2. Directions for Future Research

The CSP-i system is a full implementation of the new language defined in this thesis. On completion of the original project specifications, the author was faced with the question "Where do we go from here?".

Several minor improvements to the basic CSP notation have been suggested in the literature. Support for recursion has been proposed, as well as a method of assigning dynamic priorities to guards along

the lines of the suggestions in [Nat86].

Cousot suggests that from a mathematical point of view, it would be interesting to determine whether automatic buffering and unbounded process activation in CSP have properties simple enough to prove and which justify their inclusion in a programming language [Cou80].

Of far greater significance though, would be an implementation designed for a multi-processor architecture, with or without shared memory. Similar research is underway elsewhere [Shw78, Pat84].

In the CSP-i context, the author has considered various ways in which a distributed system may be implemented. If the processors share memory, the only change would be to the scheduler, to enable it to allocate processes to the various processors. The code for the scheduler would then have to be executed as a critical section to ensure process integrity.

In a network system without shared memory, the major change would be to add a mechanism to support interprocess communication. This involves additional code for exchanging process state information as well as for transferring actual messages.

It is the author's opinion, that either of these distributed models can be based on the CSP-i system without major revision to the current implementation.

Bibliography

BIBLIOGRAPHY

Aho74 Aho, A.V. and Johnson, S.C. LR Parsing. ACM Computing Surveys, 6(2), 99-124 (1974).

And81 Andrews, G.R. Synchronizing Resources. ACM Trans. Programming Languages and Systems, 3(4), 405-430 (1981).

And82 Andrews, G.R. The Distributed Programming Language SR - Mechanisms, Design and Implementation. Software - Practice and Experience, 12(8), 719-754 (1982).

And83 Andrews, G.R. and Schneider, F.B. Concepts and Notations for Concurrent Programming. ACM Computing Surveys, 15(1), 3-43 (1983).

Apt80 Apt, K.R., Francez, N. and De Roever, W.P. A Proof System for Communicating Sequential Processes. ACM Trans. Programming Languages and Systems, 2(3), 359-385 (1980).

Bac79 Backhouse, R.C. Syntax of Programming Languages: Theory and Practice. Prentice-Hall, Englewood Cliffs, New Jersey, (1979).

Bar83 Barter, C.J. Communications Policy for Composite Processes. The Australian Computer Journal, 15(1), 9-16 (1983).

Ben82 Ben-Ari, M. Principles of Concurrent Programming. Prentice-Hall, Englewood Cliffs, New Jersey, (1982).

Ber80 Bernstein, A.J. Output Guards and Nondeterminism in "Communicating Sequential Processes". ACM Trans. Programming Languages and Systems, 2(2), 234-238 (1980).

Bri73a Brinch Hansen, P. Concurrent Programming Concepts. ACM Computing Surveys, 5(4), 223-245 (1973).

Bri73b Brinch Hansen, P. Operating System Principles. Prentice-Hall, Englewood Cliffs, New Jersey, (1973).

Bri75 Brinch Hansen, P. The Programming Language Concurrent Pascal. IEEE Trans. Software Engineering, SE-1(2), 199-207 (1975).

Bri78 Brinch Hansen, P. Distributed Processes: A Concurrent Programming Concept. Comm. ACM, 21(11), 934-941 (1978).

Bri82 Brinch Hansen, P. Programming a Personal Computer. Prentice-Hall, Englewood Cliffs, New Jersey, (1982).

Brn82 Barnes, J.G.P. Programming in Ada. Addison-Wesley, Reading, Massachusetts, (1982).

Brr79 Barrett, W.A. and Couch, J.D. Compiler Construction: Theory and Practice. Science Research Associates, Chicago, (1979).

Buc83 Buckley, G.N. and Silberschatz, A. An Effective Implementation for the Generalized Input-Output Construct of CSP. ACM Trans. Programming Languages and Systems, 5(2), 223-235 (1983).

Bus80 Bustard, D.W. An Introduction to Pascal-Plus. In "On the Construction of Programs" edited by R.M. McKeag and A.M. MacNaghten. Cambridge University Press, Cambridge, (1980).

Cha84 Chalmers, A.G. The Monitor and Synchroniser Concept in the Programming Language CLANG. M.Sc. Thesis, Rhodes University, Grahamstown, (1984).

Cla86 Clayton, P.G. Tech. Report. Department of Computer Science, Rhodes University, Grahamstown, (1986).

Cou80 Cousot, P. and Cousot, R. Semantic Analysis of Communicating Sequential Processes. Lecture Notes in Computer Science, 85. Springer-Verlag, Berlin, (1980).

Dij75 Dijkstra, E.W. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Comm. ACM, 18(8), 453-457 (1975).

Fel79 Feldman, J. High Level Programming for Distributed Computing. Comm. ACM, 22(6), 353-359 (1979).

Fid83 Fidge, C.J. and Pascoe, R.S.V. A Comparison of the Concurrency Constructs and Module Facilities of CHILL and Ada. The Australian Computer Journal, 15(1), 17-27 (1983).

Fra79 Francez, N. et al. Semantics of Nondeterminism, Concurrency, and Communication. Journal of Computer and System Sciences, 19, 290-308 (1979).

Fra85 Francez, N. and Yemini, S.A. Symmetric Intertask Communication. ACM Trans. Programming Languages and Systems, 7(4), 622-636 (1985).

Geh82 Gehani, N. Concurrency in Ada and Multicomputers. Computer Languages, 7, 21-23 (1982).

Geh84a Gehani, N. Ada - an Advanced Introduction, including Reference Manual for the Ada Programming Language. Prentice-Hall, Englewood Cliffs, New Jersey, (1984).

Geh84b Gehani, N. Ada Concurrent Programming. Prentice-Hall, Englewood Cliffs, New Jersey, (1984).

Gen81 Gentleman, W.M. Message Passing between Sequential Processes: the Reply Primitive and the Administrator Concept. Software - Practice and Experience, 11(5), 435-466 (1981).

Gol83 Goldberg, A. and Robson, D. Smalltalk-80: the Language and its Implementation. Addison-Wesley, Reading, Massachusetts, (1983).

Hal85 Halstead Jr., R.H. Multilisp: a Language for Concurrent Symbolic Computation. ACM Trans. Programming Languages and Systems, 7(4), 501-538 (1985).

Hoa74 Hoare, C.A.R. Monitors: an Operating System Structuring Concept. Comm. ACM, 17(10), 549-557 (1974).

Hoa78 Hoare, C.A.R. Communicating Sequential Processes. Comm. ACM, 21(8), 666-677 (1978).

Hoa85 Hoare, C.A.R. Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs, New Jersey, (1985).

Inm84a INMOS Limited. IMS T424 Transputer Reference Manual. Bristol, (1984).

Inm84b INMOS Limited. Occam Evaluation Kit. Bristol, (1984).

Jon85 Jones, G. Programming in 'occam'. Programming Research Group, Oxford University, (1985).

Kat75 Katz, S. and Manna, Z. A Closer Look at Termination. Acta Informatica, 5, 333-352 (1975).

Ker86 Kerridge, J. and Simpson, D. Communicating Parallel Processes. Software - Practice and Experience, 16(1), 63-86 (1986).

Kie79 Kieburtz, R.B. and Silberschatz, A. Comments on "Communicating Sequential Processes". ACM Trans. Programming Languages and Systems, 1(2), 218-225 (1979).

Krn78 Kernighan, B.W. and Ritchie, D.M. The C Programming Language. Prentice-Hall, Englewood Cliffs, New Jersey, (1978).

Lau78 Lauer, P.E. and Shields, M.W. Abstract Specification of Resource Accessing Disciplines: Adequacy, Starvation, Priority and Interrupts. ACM Sigplan Notices, 13(13), 41-59 (1978).

Lev81 Levin, G.M. and Gries, D. A Proof Technique for Communicating Sequential Processes. Acta Informatica, 15, 291-302 (1981).

Mao80 Mao, T.W. and Yeh, R.T. Communication Port: a Language Concept for Concurrent Programming. IEEE Trans. Software Engineering, SE-6(2), 194-204 (1980).

May83 May, D. Occam. ACM Sigplan Notices, 18(4), 69-79 (1983).

Nat86 Natarajan, N. A Distributed Synchronization Scheme for Communicating Processes. The Computer Journal, 29(2), 109-117 (1986).

Pat84 Patnaik, L.M. and Badrinath, B.R. Implementation of CSP-S for Description of Distributed Algorithms. Computer Languages, 9(3/4), 193-202 (1984).

Qca84 QCAD Systems Inc. QPARSER Translator Writing System, (1984).

Rau85 Rauchle, T. and Toueg, S. Exposure to Deadlock for Communicating Processes is hard to detect. Information Processing Letters, 21, 62-68 (1985).

Rop81 Roper, T.J. and Barter, C.J. A Communicating Sequential Process Language and Implementation. Software - Practice and Experience, 11(11), 1215-1234 (1981).

Sch82 Schneider, F.B. Synchronization in Distributed Programs. ACM Trans. Programming Languages and Systems, 4(2), 125-148 (1982).

Shw78 Schwarz, J.S. Distributed Synchronization of Communicating Sequential Processes. Tech. Report, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, (1978).

Sil79 Silberschatz, A. Communication and Synchronization in Distributed Systems. IEEE Trans. Software Engineering, SE-5(6), 542-546 (1979).

Sil81 Silberschatz, A. Port Directed Communication. The Computer Journal, 24(1), 78-82 (1981).

Sme83 Smedema, C.H. et al. The Programming Languages Pascal, Modula, CHILL, Ada. Prentice-Hall, Englewood Cliffs, New Jersey, (1983).

Tsu84 Tsujino, Y. et al. Concurrent C: A Programming Language for Distributed Multiprocessor Systems. Software - Practice and Experience, 14(11), 1061-1078 (1984).

VdB80 Van den Bos, J. Comments on Ada Communication. ACM Sigplan Notices, 15(6), 77-81 (1980).

VdB81 Van den Bos, J. et al. Process Communication Based on Input Specifications. ACM Trans. Programming Languages and Systems, 3(3), 224-250 (1981).

VdS81 Van de Snepscheut, J.L.A. Synchronous Communication between Asynchronous Components. Information Processing Letters, 13(3), 127-130 (1981).

Wel80 Welsh, J., Lister, A. and Salzman, E.J. A Comparison of two Notations for Process Communication. Lecture Notes in Computer Science, 79. Springer-Verlag, Berlin, (1980).

Wel81 Welsh, J. and Lister, A. A Comparative Study of Task Communication in Ada. Software - Practice and Experience, 11(3), 257-290 (1981).

Wil84 Williamson, R. and Horowitz, E. Concurrent Communication and Synchronization Mechanisms. Software - Practice and Experience, 14(2), 135-151 (1984).

Win84 Winston, P.H. and Horn, B.K.P. LISP. Addison-Wesley, Reading, Massachusetts, (1984).

Wir71 Wirth, N. The Programming Language Pascal. Acta Informatica, 1, 35-63(1971).

Wir77 Wirth, N. Modula: a Language for Modular Multiprogramming. Software - Practice and Experience, 7(1), 3-35 (1977).

Wir83 Wirth, N. Programming in Modula-2 and Report on the Programming Language Modula-2. Springer-Verlag, Berlin, (1983).

Appendix A

```
\APPENDIX A


\CSP-i grammar -- Vers. 2.1 October 1986
\Given in the form of production rules, suitable for use by QPARSER


#WT BOOLEAN BYTE CHAR CHARCONS CONST INTEGER MESSAGE OTHERWISE
#WT REAL SIGNAL SKIP


Goal -> ConstDecl MessageDecl SigDecl StructCommand

ParCommand      -> ParCommand  //  Process         #paral
                -> Process                         #process
Process         -> ProcLabel Block CommandList     #procdecl
ProcLabel       -> _<identifier>                   #plab1
                -> _<identifier> Labhead ( Labelsubs ) #plab2
                -> <empty>                         #plab3
Labhead         -> <empty>                         #prhead
Block           -> ::                              #blok
Labelsubs       -> Labelsubs , Labsub              #labs
                -> Labsub
Labsub          -> UnsigInt                        #lab1
                -> Range                           #lab2
Range           -> <identifier> : CRange           #rdecl
CRange          -> SignIntCons .. SignIntCons      #crange


\CSP-i Commands

Command         -> SimpCommand
                -> StructCommand
SimpCommand     -> NullComm
                -> AssignComm
                -> InputComm
                -> OutputComm
                -> AltComm
                -> RepComm
StructCommand   -> [ ParCommand ]                  #structcomm
CommandList     -> VarDeclarations Commands
                -> Commands
Commands        -> Commands ; Command
                -> Command
NullComm        -> SKIP                            #nullcomm


\Assignment Commands

AssignComm      -> TargetVar := Expression         #assign
Expression      -> BoolExp
                -> StructExp                       #structexp
StructExp       -> Constructor  BoolList  }        #mesexp
                -> Constructor }                   #sigexp
Constructor     -> <identifier>  {                 #constructor
                -> {
TargetVar       -> Simpvar
                -> StructTarget
```

```
StructTarget     -> Constructor }                       #signalvar
                 -> Constructor  TargetVars }           #messagevar
TargetVars       -> TargetVars , Simpvar                #tvar1
                 -> Simpvar                             #tvar2


\Communication Commands

InputComm        -> Source ? TargetVar                  #readin
OutputComm       -> Destination ! Expression            #writeout
Source           -> ProcessName
Destination      -> ProcessName
ProcessName      -> Simpvar                             #pname


\Alternative and Repetitive Commands

RepComm          -> Repeat AltComm                      #rept
Repeat           -> *                                   #repbegin
AltComm          -> [ GCommand ]                        #alter
AltStart         -> <empty>                             #altbegin
GCommand         -> GCommand  Nextg GuardComm           #cond1
                 -> AltStart GuardComm                  #cond2
Nextg            -> []                                  #nextg
GuardComm        -> Guard Then  CommandList             #ifthen
                 -> MultiGrd Guard Then  CommandList #rangifthen
                 -> OTHERWISE Then CommandList          #defgrd
Then             -> ->                                  #then
MultiGrd         -> ( RangeList )                       #multigd
RangeList        -> RangeList , Range                   #range1
                 -> Range                               #range2
Guard            -> BoolGuard                           #bgrd
                 -> BoolGuard ; InpGuard                #bigrd
                 -> VarDeclarations InpGuard            #vigrd
                 -> InpGuard                            #igrd
InpGuard         -> InputComm
                 -> OutputComm
BoolGuard        -> BoolGuard ; OneBool                 #glist1
                 -> OneBool                             #glist2
OneBool          -> VarDeclarations  BoolExp            #vgard1
                 -> BoolExp                             #vgard2


\Declarations

VarDeclarations -> VarDeclarations  Varitem
                -> Varitem
Varitem          -> IdentList : Type ;                  #vitem
IdentList        -> IdentList , <identifier>            #idl1
                 -> <identifier>                        #idl2
Type             -> SimpleType
                 -> ArrayType
SimpleType       -> <identifier>                        #dectyp
                 -> REAL                                #rtyp
                 -> INTEGER                             #inttyp
                 -> BOOLEAN                             #btyp
                 -> BYTE                                #byttyp
                 -> CHAR                                #chtyp
```

```
ArrayType      -> [ Arraydims ] SimpleType         #arraytype
Arraydims      -> Arraydims , CRange               #adim1
               -> CRange

ConstDecl      -> CONST ConstList
               -> <empty>
ConstList      -> ConstList ConstItem
               -> ConstItem
ConstItem      -> IdentList = Constant ;           #consid

SigDecl        -> SIGNAL SigList ;
               -> <empty>
SigList        -> SigList , SigTyp
               -> SigTyp
SigTyp         -> <identifier>                     #sgltyp

MessageDecl    -> MESSAGE MessageList ;
               -> <empty>
MessageList    -> MessageList , MessageItem
               -> MessageItem
MessageItem    -> <identifier> { }                 #sigitem
               -> <identifier> { FieldList }       #mesitem
FieldList      -> FieldList , Type                 #flist1
               -> Type                             #flist2


\Expression Evaluation

BoolList       -> BoolList , BoolExp               #blist1
               -> BoolExp                          #blist2
BoolExp        -> BoolTerm
               -> BoolExp | BoolTerm               #orop
BoolTerm       -> BoolUnary
               -> BoolTerm & BoolUnary             #andop
BoolUnary      -> BoolPri
               -> ~ BoolPri                        #notop
BoolPri        -> SimpExp
               -> SimpExp Relop SimpExp            #relop
Relop          -> <                                #less
               -> >                                #grtr
               -> <=                               #lseq
               -> >=                               #gteq
               -> =                                #eqal
               -> <>                               #nteq
SimpExp        -> SimpExp + Term                   #addit
               -> SimpExp - Term                   #subt
               -> Term
Term           -> Term * Unary                     #mult
               -> Term / Unary                     #quot
               -> Term % Unary                     #modulo
               -> Unary
Unary          -> Primary
               -> - Primary                        #negate
Primary        -> ( BoolExp )                      #parexp
               -> Simpvar                          #privar
               -> UnsigValu
```

```
\General Productions

UnsigInt        -> <identifier>                    #intid
                -> <integer>
SignIntCons     -> UnsigInt
                -> + UnsigInt
                -> - UnsigInt                      #negint
UnsigValu       -> <integer>
                -> <real>
                -> <string>
                -> CHARCONS                        #chrcnst
UnsigCons       -> <identifier>                    #idval
                -> UnsigValu
Constant        -> UnsigCons
                -> + UnsigCons                     #poscons
                -> - UnsigCons                     #negcons
Simpvar         -> Varib Extension                 #varid
Varib           -> <identifier>                    #varh
Extension       -> <empty>
                -> [ BoolList ]                    #arrid
                -> ( BoolList )                    #labid
```

Appendix B

APPENDIX B

Given in this appendix, is a list of the INTERMEDIATE LANGUAGE CODES emitted by the CSP-i compiler as an interface to the interpreter.

Mnemonic	Operands	Description

1. Arithmetic and Logical Opcodes

Mnemonic	Operands	Description
ADI	<none>	Adds the integers at TOS and TOS-1
ADR	<none>	Adds the real numbers at TOS and TOS-1
SBI	<none>	Subtracts the integer at TOS from the one at TOS-1
SBR	<none>	Subtracts the real number at TOS from the one at TOS-1
MLI	<none>	Multiplies the integers at TOS and TOS-1
MLR	<none>	Multiplies the real numbers at TOS and TOS-1
DVI	<none>	Divides the integer at TOS-1 by the integer at TOS
DVR	<none>	Divides the real number at TOS-1 by the real number at TOS
MOD	<none>	Gives the modulus of the integers at TOS-1 and TOS
NOT	<none>	Gives the logical negation of the Boolean at TOS
NGI	<none>	Negates the integer at TOS
NGR	<none>	Negates the real number at TOS
EQI	<none>	Compares the integers at TOS and TOS-1 for equality
EQR	<none>	Compares the real numbers at TOS and TOS-1 for equality
GTI	<none>	Compares the integers at TOS and TOS-1 for greater than
GTR	<none>	Compares the real numbers at TOS and TOS-1 for greater than
LSI	<none>	Compares the integers at TOS and TOS-1 for less than
LSR	<none>	Compares the real numbers at TOS and TOS-1 for less than
AND	<none>	Gives the logical AND of the Boolean values at TOS and TOS-1
ORR	<none>	Gives the logical OR of the Boolean values at TOS and TOS-1

2. Conversion Opcodes

Mnemonic	Operands	Description
FLO	<none>	Converts the integer at TOS to a real number
FIX	<none>	Converts the real number at TOS to an integer

3. Input and Output Opcodes

Opcode	Operand	Description
MSI	<message format> -1	Input a message from the channel at the TOS, storing the values at the addresses given by TOS-1 ... TOS-(n/2), where n is the length of the message format
MSO	<message format> -1	Output to the channel given by TOS, the values stored at TOS-1 ... TOS-(n/2), where n is the length of the message format

4. Opcodes used in Guarded Commands

Opcode	Operand	Description
GDB	<address of EBL opcode>	Marks the start of the if-part of a guarded command
GDE	<none>	Marks the end of the if-part of a guarded command
JBL	<address of EBL opcode>	Marks the start of the jump table
EBL	<none>	Marks the end of the jump table and causes guard ring to be executed
CRG	<none>	Causes a change in the guard ring if a nested alternative command succeeds

5. Jump Opcodes

Opcode	Operand	Description
ALN	<address of JBL opcode>	Marks the start of an alternative command and causes a transfer to the address given as the operand
AJP	<address of guarded command> <dimension of range> <label$_1$>..<label$_n$>	Entry in jump table for guarded commands giving the actual values of the labels and the address of the code
AJD	<address of default guard>	Entry in jump table for the otherwise guard giving the address of the code
FJP	<destination address>	Jump to destination if TOS is false
UJP	<destination address>	Unconditional jump to destination
EDF	<destination address>	Opcode executed at the end of the default guard. If no interrupt has occurred, jump to the destination address, else execute the guarded command wishing to establish communication.

6. Load and Store Opcodes

Opcode	Operand	Description
LDI	<level> <offset>	Load integer at address given by display level + offset onto TOS
LDR	<level> <offset>	Load real number at address given by display level + offset onto TOS

Opcode	Operands	Description
LDA	<size> <level> <offset>	Load size number of bytes, starting at address given by display level and offset, on TOS
LOI	<none>	Load integer at address given by TOS onto TOS
LOR	<none>	Load real number at address given by TOS onto TOS
LOA	<size>	Load onto TOS, <size> number of bytes starting at address given by TOS
LDS	<n> <ch$_1$> .. <ch$_n$>	Load the n characters given as operands onto TOS
LSG	<none>	Check for a signal -- this opcode cannot appear in a program that compiles successfully
LCH	<process num> <dimension>	Load process number on TOS and number of dimensions in process label at TOS-1
STI	<level> <offset>	Store the integer on TOS at the address given by level and offset
STR	<level> <offset>	Store the real number on TOS at the address given by display level and offset
STA	<size> <level> <offset>	Store size number of bytes found at TOS ... TOS-size+1 at the address given by display level and offset
SOI	<none>	Store the integer at TOS-1 at the address given by TOS
SOR	<none>	Store the real number at TOS-1 at the address given by TOS
SOA	<size>	Store the values at TOS-1 .. TOS-size at the locations starting at the address given by TOS
LID	<level> <offset>	Load onto TOS the address defined by display level + offset
IND	<lower> <upper> <size>	Leave at TOS the address formed by adding the subscript given at TOS to the address given at TOS-1
PSI	<value>	Load an integer value on TOS
PSR	<value>	Load a real value on TOS

7. Miscellaneous Opcodes

Opcode	Operands	Description
INC	<integer>	Increment the stack pointer by value of operand
DEC	<integer>	Decrement the stack pointer by value of operand
BEG	<process number> <runtime level> <parent process> <instance dimension>	Marks the beginning of a process and allows a process descriptor to be constructed
EXT	<none>	Marks the end of a process
HLT	-1	Final opcode in CSP-i program
NOP	<none>	No operation

8. Debugging Opcodes

STK	<none>	Dump the contents of the stack frame of the current process
DMP	<none>	List the process descriptor ring

9. Preprocessor Opcodes

LOC <label>	Associate the label operand with the current code address
DEF <label> <value>	Assign the value to the label
LAB <low> <high>	Follows the BEG opcode giving the range (low..high) of a single dimension of the process label

Appendix C

APPENDIX C

A few sample CSP-i programs are given in this appendix to illustrate the syntax of the language and its suitability to most concurrent algorithms.

```
\ Example 1 \
\ Parallel Sieve of Eratosthenes based on the solution given in
  [Hoa78]. 25 processes are used to print the first 25 prime numbers
  less than 1000. \

const LIM = 1000;
      PROS = 25;

 [ _sieve(0) ::
       n : integer;

     screen ! 2; n := 3;
     *[n < LIM -> sieve(1) ! n; n := n + 2]

 //

   _sieve(26) ::

    *[n : integer; sieve(PROS) ? n -> screen ! n]

 //

   _sieve(i:1 .. PROS) ::
      p, mp : integer;

    sieve(i-1) ? p; screen ! p;

    mp := p;  \mp is a multiple of p \
    *[m :integer; sieve(i-1) ? m ->

        *[m > mp -> mp := mp + p];
        [ m = mp -> skip
         []
         m < mp -> sieve(i+1) ! m
        ]
     ]

 ]
```

```
\ Example 2 \
\ CSP-i program to simulate the vending machine described in [Hoa85].
  By pressing the character 'c', followed by an integer value, the
  user may insert a coin.  Goods are requested by entering a 'p'.
  This program halts if a 'q' is pressed. \

const PRICE = 20;
message insert{integer}, deliver{integer,integer};

 [ _machine ::
    items, paid : integer;
    items := 5; paid := 0;

      *[ coin : integer; user ? insert{coin} -> paid := paid+coin
           []
         user ? push{} ->
                [items > 0; paid >= PRICE ->
                              user !deliver{paid-PRICE, 1};
                              items := items - 1;
                              paid := paid - PRICE

                       []
                 items = 0 | paid < PRICE -> user ! deliver{paid, 0}
                ]
       ]

  //

  _user ::
    in : char; terminate : boolean;

    terminate := false;

   *[ ~terminate; kbd ? in ->
               [ coin : integer; in = 'c'  -> kbd ? coin;
                                              machine ! insert{coin};
                                              screen ! {'C', coin}
                  []
                left, units : integer; in = 'p' ->
                                    machine ! push{};
                                    machine ? deliver {left,units};
                                    screen !  {'D', units, 'L', left}
                  []
                in = 'q' -> terminate := true
                  []
                otherwise -> skip  \prevents program from aborting if
                                    incorrect key is pressed\
               ]
     ]
 ]
```

```
\ Example 3 \
\ Conway's problem -- where 10-column cards are read and re-written
  as 15-column lines.  After every card is read an extra blank is
  inserted, and a pair of asterisks ** is replaced by an arrow ^.
  [Ben82] \

const BBUF = 19;
      LIM  = 20;     \ Buffer size \
      CLIM = 10;     \ Input card size \
      PLIM = 15;     \ Output line length \

[ _buf ::
     buffer : [0 .. BBUF] char;
     in, out : integer;
     c,ch : char;

      in := 0; out := 0;
      *[ in < out + LIM; card ? c ->
                  [c = '*'; card ? ch ->
                      [ ch = '*' -> buffer[in % LIM] := '^';
                                    in := in + 1
                       []
                        ch <> '*' -> buffer[in % LIM] := c;
                                     buffer[(in+1) % LIM] := ch;
                                     in := in + 2
                       ]

                   []
                    c <> '*' -> buffer[in % LIM] := c;
                                in := in + 1
                  ]
         []
         out < in; print ! buffer[out % LIM] -> out := out + 1
      ]

   //

  _card ::
     ch : char; x : integer;

      x := 0;
      kbd ? ch;

     *[ ch <> '#' ->
             [ x = CLIM -> buf ! ' '; x := 0
                 []
               x <> CLIM -> buf ! ch; x := x + 1
             ];
             kbd ? ch
      ]

   //
```

```
_print ::
   c, ret : char; i : integer;

   i := Ø; ret := 13;

  *[ buf ? c -> [ i = PLIM -> screen ! ret; i := Ø
                 []
                  otherwise -> skip
                ];
                screen ! c; i := i + 1
   ]

]
```

```
\ Example 4 \
\ Matrix Multiplication: A1 x A2 multiplied by a B1 x B2 matrix
  to give a B2 x A2 matrix.  No provision is made to change the
  values of the A matrix, or to terminate the program.  The
  elements of the B matrix are incremented by 1 for the next
  iteration. \

const A1 = 3;        \ A row -- must match B column\
      A2 = 4;        \ A column \
      A1PLUS = 4;
      A2PLUS = 5;
      B1 = 3;        \ B column \
      B2 = 2;        \ B row \

[ _M (i: 1 .. A1, Ø, k: 1 .. B2) ::
  \ process WEST \

  B : [1 .. B1, 1 .. B2] integer;

  B[i,k] := i+k;
  * [ M (i, 1, k) ! B[i,k] -> B[i,k] := B[i,k] + 1 ]

  //

  _M (Ø, j: 1 .. A2, k: 1 .. B2) ::
  \ process NORTH \

  *[ M (1, j, k) ! Ø  -> skip ]

 //

  _M (i: 1 .. A1, A2PLUS, k: 1 .. B2) ::
  \ process EAST \

  *[ x : integer; M(i,A2,k) ? x -> skip ]

 //

  _M (A1PLUS, j: 1 .. A2, k: 1 .. B2) ::
  \ process SOUTH \

  *[ res : integer; M(A1,j,k) ? res -> screen ! {j,k,res} ]

 //

  _M (i: 1 .. A1, j: 1 .. A2, k: 1 .. B2) ::
  \process CENTER \

   A : [1 .. A1, 1 .. A2] integer;

   A[i,j] := i+j;
  *[ x,sum : integer; M(i,j-1,k) ? x ->
                          M(i,j+1,k) ! x;
                          M(i-1, j,k) ? sum;
                          M(i+1, j,k) ! (A[i,j] * x + sum)
   ]
 ]
```

Appendix D & E

APPENDIX D

```
/* CSP-i parser definitions */

#include <a:ctype.h>
#include <a:cspio.h>

#define READ_MODE    "r"
#define WRITE_MODE   "w"
#define MAXINT       32767

#define STACKSIZE 60       /* maximum size of LR(1) parser stack */
#define EOS '\0'           /* marks end of string */
#define EOLCH '\n'         /* end of line character */
#define EOF  '\x0C'        /* dummy eof character  -- form feed */
#define LINELEN 80         /* maximum length of a line */
#define STRTABLEN 500      /* maximum number of chars in string table */
#define STRING_QUOTE '"'   /* character delimiting quoted strings */
#define CHAR_QUOTE '\''    /* character delimiting character constants */
#define DBUGCHAR '$'       /* character to initiate debugging procedures */
#define PRDUMP '^'         /* emits the opcode to initiate a process dump */
#define STACDUMP '@'       /* emits the opcode for a stack dump */
#define MAXERRORS 40       /* maximum errors before aborting */
#define HASHSIZE 67        /* hash table size -- prime number! */
#define HLIMIT 66          /* limit in hash table (hashsize minus one) */
#define MAXTOKLEN 15       /* length of a token or symbol */
#define ERRSYMLEN 7        /* length of err#xx plus a terminator */
#define NORESERVED 10      /* no. of reserved keywords */

#define LABELMAX 100     /* maximum number of dummy labels used (0 - 99) */
#define MAXMESS 20       /* (max no.* 2) of i/o variables/expressions */
#define MAXDIM 5         /* max no. of ranges for a guard */
#define FIRSTADR 2       /* first offset for local vars; 0=static, 1=ret adr */
#define FIRSTPROC 2      /* first process number for user processes; */
                         /* 0=input,1=output */
#define GLOBLEV 0        /* global level used for sym entries and plevel */
#define NOSTRUCT 0       /* used in I/O formats if empty constructor used */

#define ID_TOKLEN 15      /* maximum user identifier length */
#define MAXRPLEN 6        /* length of longest production right part */
#define TERM_TOKS 50      /* number of terminal tokens */
#define NTERM_TOKS 77     /* number of nonterminal tokens */
#define ALL_TOKS 127      /* term_toks + nterm_toks */
#define IDENT_TOKX 23     /* token number of <identifier> */
#define INT_TOKX 24       /* token number of <integer> */
#define REAL_TOKX 25      /* token number of <real> */
#define STR_TOKX 27       /* token number of <string> */
#define STOP_TOKX 26      /* token number of stopsign (end-of-file) */
#define GOAL_TOKX 77      /* token number of goal */
#define EOL_TOKX 22       /* token number of end-of-line */
#define READSTATE 137     /* first READ state */
#define LOOKSTATE 245     /* first LOOK state */
#define MAXSTATE 276      /* largest state number */
#define REDUCELEN 136     /* number of productions */
#define RLTOKENS 345
#define SSTOKENS 217
#define PRODTOKS 489
```

```
#define TOKCHARS 915
#define START_STATE 245  /* initial state */
#define STK_STATE_1 223  /* state initially pushed on stack */
#define ADDIT 1
#define ADIM1 2
#define ALTBEGIN 3
#define ALTER 4
#define ANDOP 5
#define ARRAYTYPE 6
#define ARRID 7
#define ASSIGN 8
#define BGRD 9
#define BIGRD 10
#define BLIST1 11
#define BLIST2 12
#define BLOK 13
#define BTYP 14
#define BYTTYP 15
#define CHRCNST 16
#define CHTYP 17
#define COND1 18
#define COND2 19
#define CONSID 20
#define CONSTRUCTOR 21
#define CRANGE 22
#define DECTYP 23
#define DEFGRD 24
#define EQAL 25
#define FLIST1 26
#define FLIST2 27
#define GLIST1 28
#define GLIST2 29
#define GRTR 30
#define GTEQ 31
#define IDL1 32
#define IDL2 33
#define IDVAL 34
#define IFTHEN 35
#define IGRD 36
#define INTID 37
#define INTTYP 38
#define LAB1 39
#define LAB2 40
#define LABID 41
#define LABS 42
#define LESS 43
#define LSEQ 44
#define MESEXP 45
#define MESITEM 46
#define MESSAGEVAR 47
#define MODULO 48
#define MULT 49
#define MULTIGD 50
#define NEGATE 51
#define NEGCONS 52
#define NEGINT 53
#define NEXTG 54
#define NOTOP 55
```

```
#define NTEQ 56
#define NULLCOMM 57
#define OROP 58
#define PARAL 59
#define PAREXP 60
#define PLAB1 61
#define PLAB2 62
#define PLAB3 63
#define PNAME 64
#define POSCONS 65
#define PRHEAD 66
#define PRIVAR 67
#define PROCDECL 68
#define PROCESS 69
#define QUOT 70
#define RANGE1 71
#define RANGE2 72
#define RANGIFTHEN 73
#define RDECL 74
#define READIN 75
#define RELOP 76
#define REPBEGIN 77
#define REPT 78
#define RTYP 79
#define SGLTYP 80
#define SIGEXP 81
#define SIGITEM 82
#define SIGNALVAR 83
#define STRUCTCOMM 84
#define STRUCTEXP 85
#define SUBT 86
#define THEN 87
#define TVAR1 88
#define TVAR2 89
#define VARH 90
#define VARID 91
#define VGARD1 92
#define VGARD2 93
#define VIGRD 94
#define VITEM 95
#define WRITEOUT 96
#define BOOLEAN 32
#define BYTE 33
#define CHAR 34
#define CHARCONS 35
#define CONST 36
#define INTEGER 37
#define MESSAGE 38
#define OTHERWISE 39
#define REAL 40
#define SIGNAL 41
#define SKIP 42

/* Enumerated values for SYMTYPE */
/* NB. changes here must also be reflected in the symtypename array in
   CSPDECLS.C */

#define RESERVED 0     /* input, output, true, false */
```

```
#define SYMERR   1
#define USER     2
#define VARI     3
#define CONSVAR  4
#define INT_TYPE 5
#define REAL_TYPE 6
#define BOOL_TYPE 7
#define USER_TYPE 8
#define PROC_TYPE 9
#define ARRAY_TYPE 10
#define MESS_TYPE 11
#define SUBRANGE 12
#define SIG_TYPE 13
#define CHAR_TYPE 14   /* includes bytes */
#define UNK_TYPE 15

#define LAST_SYMTYPE  15


/* Enumerated values for SEMTYPE */
/* NB. changes here must also be reflected in the semtypename array in
   CSPDECLS.C */

#define NONE  0
#define IDENT  1
#define STRNG  2
#define TYP_NODE  3
#define IDLIST_NODE  4
#define AR_EXT  5

#define FIXED  6      /* integer, char and byte */
#define FLT  7
#define ADD_NODE  8
#define SUB_NODE  9
#define MPY_NODE  10
#define QUOT_NODE  11
#define MOD_NODE  12
#define NEG_NODE  13

#define BOOLS  14
#define AND_NODE  15
#define OR_NODE  16
#define NOT_NODE  17
#define EQ_NODE  18
#define GEQ_NODE  19
#define GTR_NODE  20
#define LEQ_NODE  21
#define LES_NODE  22
#define NEQ_NODE  23

#define BLIST_NODE  24
#define JP_NODE  25
#define REP_NODE  26
#define MESS_NODE  27
#define SIG_NODE  28
#define MES_EXT  29
#define PROC_EXT  30
#define ALT_NODE  31
```

```
#define SEM_ERR  32          /* must be = last_semtype */
#define LAST_SEMTYPE  32

/* define number of words in types  */
#define INT_SIZE 1           /* 2 bytes */
#define BOOL_SIZE 1
#define CHAR_SIZE 1
#define REAL_SIZE 4          /* size of double = 8 bytes */
#define SIG_SIZE 0

typedef short int small_int;
typedef short int symtype;
typedef short int semtype;

/* Structure for symbol table entries - identifiers and types */

typedef struct symtabtype
   {
     struct symtabtype *next;
     int level;                     /* compile-time level */
     int rlevel;                    /* runtime level */
     char sym[MAXTOKLEN+1];
     symtype symt;                  /* values: reserved..user */
     union {
         struct {                   /* symt = CONSVAR  */
                  semtype constyp; /* type of constant */
                  int consival;
                  float consrval;
                } csymt;
         int tokval;                /* symt = RESERVED */
         struct {                   /* symt = VARI     */
                  int saddr;        /* base address */
                  boolean canchange;           /* distinguishes label vars */
                  struct symtabtype *vtypep,   /* its type */
                                    *nxtvar;   /* used by label+mess vars */
                } vsymt;
         struct symtabtype *utypep; /* symt = USER_TYPE */
         struct {                   /* symt = std. scalar types, ARRAY_TYPE,
                                       MESS_TYPE, SIG_TYPE and SUBRANGE */
                   struct symtabtype *subtype;   /* its type */
                   int size, udim, ldim;         /* dimensions */
                } asymt;
         struct {                   /* symt = PROC_TYPE */
                   int procnum,     /* number of first instance of process */
                       noprocs,     /* total no. of instances of process */
                       locals,      /* no. of local vars */
                       dim;         /* dimension of label */
                   boolean defined;
                   struct symtabtype
                           *labptr,  /* identifiers in label */
                           *nxtproc, /* next process with same name */
                           *parproc; /* previous process */
                } psymt;

        }  usym;
     } *symtabp;
```

```
/* Structure for semantic stack  */

typedef struct semrec
    {
      semtype semt;              /* values: other..strng  */
      symtabp svtype;            /* type pointer where appropriate */
      union {
         struct { symtabp symp;  /* semt = IDENT */
                     struct semrec *vext;
                    } idsemt;
         int numval;             /* semt = FIXED, BOOLS */
         float rval;             /* semt = FLT */
         int stx;                /* semt = STRNG; (pos in strtab) */
         struct {                /* semt = AR_EXT or PROC_EXT     */
                 symtabp arsymp;          /* symbol table entry */
                 struct semrec *arextnext, /* next component */
                               *ax;        /* array index */
               } arsemt;
         int offset;             /* semt = MES_EXT */
             symtabp the_typep;  /* semt = TYPE_NODE */
         struct {                /* semt = IDLIST_NODE */
                 symtabp idsymp;        /* the identifier */
                 struct semrec *idnext; /* next idlist node */
               } insemt;
         struct {                /* semt = BLIST_NODE, that is a list
                                    of unevaluated expressions */
                 struct semrec *blnext,  /* links next blist node */
                               *blexp;   /* the expression */
               } bsemt;
         struct {                 /* semt = ADD_NODE .. NEQ_NODE */
                 struct semrec *treel, *treer;  /* binary trees */
               } msemt;
         struct semrec *treeun;   /* sqemp = NEG_NODE, NOT_NODE */
         struct {                 /* semt = MES_NODE */
                  symtabp messvar;
                  struct semrec *messexp;  /* points at a blist node */
               } gsemt;
         struct {                  /* semt = SIG_NODE */
                  symtabp sigvar;
               } sigsemt;
         struct {                  /* semt = JP_NODE or REP_NODE */
                  int jplabl, jplab2;
               } jsemt;
         struct {                  /* semt = ALT_NODE */
                  int altlabl, altlab2, altlab3, altlab4;
                  int saveadr;       /* saves address offset before alt */
                  boolean usable;    /* whether normal guard or otherwise */
                  struct semrec *altext;  /* gives range */
               } altsemt;

        } usem;
   } *semrecp;
```

```
/* Contains declarations for the CSP-i parser */

#include "CSPDEFS.C"

 /* Dynamic parser data structures */
    int stack[STACKSIZE+1];          /* the LR(1) state stack */
    semrecp semstack[STACKSIZE+1];   /* semantics stack */
    int stackx;                      /* index of top of stack */

 /* Global parser data structures (parser tables) */
  /* statex array */
   int statex[] = {
      /* 0  : */  0, 0, 1, 2, 5, 5, 5, 5, 15, 18,
      /* 10 : */  19, 21, 22, 22, 23, 25, 25, 25, 25, 25,
      /* 20 : */  25, 29, 30, 32, 33, 34, 35, 41, 30, 45,
      /* 30 : */  46, 47, 51, 51, 29, 53, 54, 57, 60, 34,
      /* 40 : */  62, 65, 68, 69, 72, 73, 74, 54, 76, 91,
      /* 50 : */  96, 96, 98, 100, 101, 102, 103, 107, 109, 116,
      /* 60 : */  124, 74, 132, 135, 137, 138, 141, 98, 142, 145,
      /* 70 : */  145, 145, 145, 145, 145, 146, 101, 152, 73, 153,
      /* 80 : */  153, 154, 162, 165, 166, 109, 168, 98, 100, 169,
      /* 90 : */  171, 171, 171, 166, 152, 141, 166, 174, 175, 176,
      /* 100: */  177, 178, 179, 137, 180, 181, 182, 183, 184, 185,
      /* 110: */  186, 187, 72, 188, 189, 193, 189, 194, 186, 195,
      /* 120: */  98, 196, 198, 209, 187, 72, 213, 72, 107, 215,
      /* 130: */  72, 196, 198, 209, 213, 213, 215, 0, 2, 5,
      /* 140: */  7, 9, 17, 19, 22, 24, 26, 26, 32, 34,
      /* 150: */  5, 43, 46, 51, 22, 54, 56, 58, 61, 61,
      /* 160: */  64, 66, 69, 77, 46, 83, 46, 90, 92, 94,
      /* 170: */  96, 98, 101, 103, 77, 106, 109, 119, 122, 132,
      /* 180: */  135, 138, 138, 122, 147, 69, 153, 156, 158, 167,
      /* 190: */  167, 176, 184, 156, 109, 191, 138, 201, 203, 206,
      /* 200: */  209, 46, 106, 77, 211, 214, 156, 167, 138, 138,
      /* 210: */  176, 176, 176, 176, 176, 176, 77, 217, 138, 220,
      /* 220: */  77, 167, 222, 224, 226, 0, 228, 230, 232, 230,
      /* 230: */  230, 232, 232, 233, 235, 238, 232, 232, 240, 242,
      /* 240: */  244, 246, 232, 255, 259, 262, 264, 266, 268, 270,
      /* 250: */  272, 274, 276, 279, 281, 283, 287, 290, 292, 294,
      /* 260: */  296, 299, 301, 303, 305, 314, 317, 321, 323, 325,
      /* 270: */  327, 329, 331, 333, 336, 340, 344};
  /* map array */
   int map[] = {
      /* 0  : */  0, 33, 0, 32, 16, 0, 0, 0, 34, 0,
      /* 10 : */  80, 0, 52, 65, 20, 17, 15, 14, 38, 79,
      /* 20 : */  23, 27, 82, 0, 0, 69, 0, 37, 46, 0,
      /* 30 : */  13, 84, 53, 0, 26, 77, 0, 57, 68, 59,
      /* 40 : */  22, 6, 2, 78, 90, 89, 83, 21, 91, 0,
      /* 50 : */  40, 39, 36, 29, 19, 54, 4, 85, 67, 96,
      /* 60 : */  75, 47, 8, 0, 62, 87, 72, 94, 55, 56,
      /* 70 : */  25, 31, 44, 30, 43, 51, 18, 81, 88, 41,
      /* 80 : */  7, 95, 74, 42, 24, 60, 50, 10, 28, 5,
      /* 90 : */  49, 70, 48, 35, 45, 71, 73, 0, 0, 0,
      /* 100: */  0, 63, 66, 61, 63, 3, 63, 0, 0, 0,
      /* 110: */  64, 33, 90, 0, 0, 3, 0, 37, 64, 67,
      /* 120: */  9, 93, 0, 0, 33, 90, 0, 90, 0, 12,
      /* 130: */  90, 92, 58, 76, 1, 86, 11};
  /* popno array */
   short int popno[] = {
```

```
    /* 0  : */  0, 1, 2, 3, 1, 1, 1, 1, 1, 3,
    /* 10 : */  1, 4, 2, 2, 4, 1, 1, 1, 1, 1,
    /* 20 : */  1, 1, 3, 3, 3, 1, 1, 1, 4, 3,
    /* 30 : */  1, 3, 2, 2, 3, 1, 1, 1, 3, 3,
    /* 40 : */  3, 4, 3, 2, 1, 1, 2, 2, 2, 2,
    /* 50 : */  1, 1, 1, 1, 2, 1, 3, 1, 1, 3,
    /* 60 : */  3, 3, 3, 3, 6, 1, 1, 2, 2, 1,
    /* 70 : */  1, 1, 1, 1, 1, 2, 3, 2, 3, 3,
    /* 80 : */  3, 4, 3, 3, 3, 3, 3, 3, 3, 3,
    /* 90 : */  3, 3, 3, 3, 3, 3, 4, 0, 0, 2,
    /* 100: */  0, 0, 0, 2, 0, 0, 0, 1, 1, 1,
    /* 110: */  1, 1, 1, 0, 1, 0, 2, 1, 1, 1,
    /* 120: */  1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    /* 130: */  1, 2, 3, 3, 3, 3, 3};
/* stk_state array */
 int stk_state[] = {
    /* 0  : */  0, 0, 225, 137, 0, 141, 146, 147, 192, 211,
    /* 10 : */  212, 213, 214, 215, 0, 146, 147, 0, 0, 154,
    /* 20 : */  0, 0, 0, 225, 0, 162, 165, 185, 0, 0,
    /* 30 : */  150, 0, 0, 0, 0, 158, 159, 164, 202, 175,
    /* 40 : */  0, 158, 159, 164, 0, 0, 0, 145, 174, 184,
    /* 50 : */  0, 164, 0, 0, 183, 178, 0, 174, 184, 0,
    /* 60 : */  229, 0, 166, 201, 0, 162, 185, 0, 0, 174,
    /* 70 : */  184, 0, 0, 0, 179, 0, 220, 216, 203, 184,
    /* 80 : */  174, 163, 171, 207, 194, 190, 189, 176, 179, 197,
    /* 90 : */  0, 194, 189, 176, 207, 0, 202, 0, 189, 0,
    /* 100: */  0, 0, 0, 168, 174, 184, 0, 183, 0, 192,
    /* 110: */  211, 212, 213, 214, 215, 0, 174, 194, 189, 176,
    /* 120: */  184, 190, 207, 0, 174, 194, 189, 176, 184, 190,
    /* 130: */  207, 0, 174, 184, 0, 174, 0, 0, 193, 206,
    /* 140: */  0, 0, 208, 209, 0, 0, 211, 212, 213, 214,
    /* 150: */  215, 0, 0, 0, 221, 190, 174, 194, 189, 176,
    /* 160: */  207, 0, 188, 219, 0, 0, 194, 0, 0, 208,
    /* 170: */  0, 211, 212, 0, 223, 224, 0, 226, 227, 228,
    /* 180: */  229, 230, 230, 0, 0, 0, 0, 0, 234, 203,
    /* 190: */  216, 220, 0, 236, 0, 0, 207, 0, 183, 178,
    /* 200: */  195, 182, 181, 196, 188, 221, 190, 218, 0, 191,
    /* 210: */  208, 209, 0, 210, 0, 182, 195, 0};
/* stk_tostate array */
 int stk_tostate[] = {
    /* 0  : */  138, 247, 138, 138, 173, 148, 12, 13, 75, 274,
    /* 10 : */  275, 90, 91, 92, 266, 12, 13, 148, 248, 29,
    /* 20 : */  151, 0, 148, 2, 247, 34, 41, 200, 21, 153,
    /* 30 : */  23, 143, 143, 145, 157, 32, 33, 40, 51, 51,
    /* 40 : */  160, 32, 33, 40, 160, 151, 163, 11, 258, 63,
    /* 50 : */  257, 40, 160, 168, 195, 195, 171, 258, 63, 257,
    /* 60 : */  39, 25, 42, 82, 161, 34, 200, 21, 161, 258,
    /* 70 : */  63, 257, 256, 180, 60, 172, 254, 254, 254, 254,
    /* 80 : */  254, 254, 45, 260, 260, 260, 260, 260, 60, 78,
    /* 90 : */  58, 190, 190, 190, 221, 174, 83, 186, 206, 193,
    /* 100: */  261, 177, 194, 43, 258, 63, 257, 62, 59, 75,
    /* 110: */  274, 275, 90, 91, 92, 266, 258, 52, 52, 52,
    /* 120: */  63, 67, 87, 257, 258, 52, 52, 52, 63, 67,
    /* 130: */  87, 257, 258, 63, 257, 258, 257, 156, 216, 220,
    /* 140: */  203, 205, 272, 89, 263, 210, 274, 275, 90, 91,
    /* 150: */  92, 266, 57, 48, 49, 49, 49, 190, 190, 190,
    /* 160: */  221, 174, 66, 95, 50, 186, 76, 54, 189, 272,
    /* 170: */  263, 274, 275, 266, 246, 248, 246, 145, 156, 167,
```

```
    /* 180: */  156, 176, 156, 170, 169, 172, 253, 173, 48, 84,
    /* 190: */  93, 96, 38, 176, 51, 266, 88, 53, 268, 268,
    /* 200: */  269, 269, 269, 204, 204, 271, 271, 276, 262, 68,
    /* 210: */  272, 89, 263, 273, 264, 199, 217, 198};
/* toknum array */
 int toknum[] = {
    /* 0  : */  23, 0, 8, 28, 0, 23, 0, 23, 0, 7,
    /* 10 : */  9, 23, 24, 25, 27, 35, 0, 47, 0, 8,
    /* 20 : */  17, 0, 23, 0, 43, 0, 23, 24, 25, 27,
    /* 30 : */  35, 0, 17, 0, 23, 32, 33, 34, 37, 40,
    /* 40 : */  43, 49, 0, 8, 17, 0, 7, 9, 23, 24,
    /* 50 : */  0, 8, 49, 0, 23, 0, 15, 0, 13, 45,
    /* 60 : */  0, 23, 24, 0, 11, 0, 8, 45, 0, 23,
    /* 70 : */  32, 33, 34, 37, 40, 43, 0, 6, 23, 42,
    /* 80 : */  43, 47, 0, 23, 32, 33, 34, 37, 40, 0,
    /* 90 : */  4, 0, 43, 0, 1, 0, 31, 0, 23, 49,
    /* 100: */  0, 16, 0, 8, 14, 0, 23, 24, 0, 4,
    /* 110: */  9, 23, 24, 25, 27, 35, 39, 50, 0, 44,
    /* 120: */  45, 0, 4, 9, 23, 24, 25, 27, 35, 47,
    /* 130: */  50, 0, 23, 47, 0, 8, 49, 0, 4, 9,
    /* 140: */  23, 24, 25, 27, 35, 50, 0, 6, 23, 42,
    /* 150: */  43, 47, 0, 5, 8, 0, 10, 0, 4, 9,
    /* 160: */  23, 24, 25, 27, 35, 50, 0, 4, 9, 23,
    /* 170: */  24, 25, 27, 35, 50, 0, 4, 9, 23, 24,
    /* 180: */  25, 27, 35, 0, 4, 23, 24, 25, 27, 35,
    /* 190: */  0, 4, 9, 23, 24, 25, 27, 35, 49, 50,
    /* 200: */  0, 23, 0, 5, 8, 0, 8, 45, 0, 17,
    /* 210: */  0, 5, 48, 0, 5, 8, 0, 8, 49, 0,
    /* 220: */  23, 0, 14, 0, 36, 0, 38, 0, 41, 0,
    /* 230: */  46, 0, 0, 47, 0, 4, 43, 0, 17, 0,
    /* 240: */  17, 0, 48, 0, 3, 0, 7, 9, 18, 19,
    /* 250: */  20, 28, 29, 30, 0, 2, 6, 12, 0, 7,
    /* 260: */  9, 0, 36, 0, 38, 0, 23, 0, 43, 0,
    /* 270: */  15, 0, 15, 0, 15, 0, 46, 15, 0, 1,
    /* 280: */  0, 16, 0, 47, 8, 14, 0, 4, 43, 0,
    /* 290: */  17, 0, 17, 0, 14, 0, 1, 31, 0, 10,
    /* 300: */  0, 48, 0, 3, 0, 7, 9, 18, 19, 20,
    /* 310: */  28, 29, 30, 0, 8, 14, 0, 2, 6, 12,
    /* 320: */  0, 47, 0, 48, 0, 48, 0, 14, 0, 48,
    /* 330: */  0, 3, 0, 7, 9, 0, 2, 6, 12, 0,
    /* 340: */  2, 6, 12, 0, 48, 0};
/* tostate array */
 int tostate[] = {
    /* 0  : */  1, 0, 140, 141, 0, 142, 0, 3, 0, 147,
    /* 10 : */  146, 8, 7, 6, 5, 4, 0, 149, 0, 150,
    /* 20 : */  9, 0, 10, 0, 249, 0, 8, 7, 6, 5,
    /* 30 : */  4, 0, 14, 0, 20, 17, 16, 15, 18, 19,
    /* 40 : */  152, 22, 0, 154, 24, 0, 159, 158, 27, 26,
    /* 50 : */  0, 162, 28, 0, 250, 0, 30, 0, 251, 31,
    /* 60 : */  0, 27, 26, 0, 164, 0, 166, 165, 0, 20,
    /* 70 : */  17, 16, 15, 18, 19, 152, 0, 35, 255, 37,
    /* 80 : */  252, 36, 0, 20, 17, 16, 15, 18, 19, 0,
    /* 90 : */  175, 0, 115, 0, 178, 0, 179, 0, 44, 46,
    /* 100: */  0, 183, 0, 140, 185, 0, 259, 26, 0, 188,
    /* 110: */  192, 265, 7, 6, 5, 4, 187, 191, 0, 55,
    /* 120: */  56, 0, 196, 192, 267, 7, 6, 5, 4, 36,
    /* 130: */  191, 0, 267, 36, 0, 197, 61, 0, 196, 192,
    /* 140: */  44, 7, 6, 5, 4, 191, 0, 35, 267, 37,
```

```
    /* 150: */  252, 36, 0, 64, 202, 0, 65, 0, 196, 192,
    /* 160: */  270, 7, 6, 5, 4, 191, 0, 196, 192, 265,
    /* 170: */  7, 6, 5, 4, 191, 0, 196, 192, 44, 7,
    /* 180: */  6, 5, 4, 0, 196, 44, 7, 6, 5, 4,
    /* 190: */  0, 196, 192, 44, 7, 6, 5, 4, 77, 191,
    /* 200: */  0, 44, 0, 79, 218, 0, 218, 80, 0, 81,
    /* 210: */  0, 85, 208, 0, 86, 219, 0, 218, 94, 0,
    /* 220: */  222, 0, 201, 0, 137, 0, 139, 0, 144, 0,
    /* 230: */  155, 0, 0, 47, 0, 181, 182, 0, 184, 0,
    /* 240: */  207, 0, 208, 0, 209, 0, 211, 212, 74, 72,
    /* 250: */  69, 70, 73, 71, 0, 215, 213, 214, 0, 211,
    /* 260: */  212, 0, 223, 97, 224, 98, 225, 99, 100, 226,
    /* 270: */  101, 227, 103, 102, 104, 229, 230, 106, 105, 108,
    /* 280: */  107, 109, 110, 233, 111, 111, 112, 234, 234, 113,
    /* 290: */  235, 114, 235, 116, 222, 117, 118, 118, 119, 120,
    /* 300: */  238, 239, 121, 240, 122, 241, 241, 241, 241, 241,
    /* 310: */  241, 241, 241, 123, 124, 124, 125, 243, 243, 243,
    /* 320: */  126, 233, 127, 239, 128, 239, 129, 222, 130, 239,
    /* 330: */  131, 240, 132, 244, 244, 133, 243, 243, 243, 134,
    /* 340: */  243, 243, 243, 135, 239, 136};
/* insym array */
 int insym[] = {
    /* 0  : */  0, 23, 68, 23, 35, 27, 25, 24, 23, 17,
    /* 10 : */  23, 113, 124, 124, 17, 34, 33, 32, 37, 40,
    /* 20 : */  23, 53, 49, 87, 17, 97, 24, 23, 49, 106,
    /* 30 : */  15, 45, 123, 123, 53, 6, 47, 42, 65, 97,
    /* 40 : */  123, 110, 63, 51, 23, 111, 49, 47, 74, 127,
    /* 50 : */  99, 123, 93, 92, 79, 44, 45, 114, 111, 73,
    /* 60 : */  115, 49, 73, 113, 5, 10, 99, 93, 60, 20,
    /* 70 : */  28, 30, 19, 29, 18, 124, 79, 49, 111, 5,
    /* 80 : */  45, 17, 63, 85, 65, 5, 5, 93, 92, 60,
    /* 90 : */  124, 124, 124, 65, 49, 99, 65, 77, 67, 68,
    /* 100: */  86, 43, 23, 23, 13, 43, 43, 98, 98, 111,
    /* 110: */  111, 23, 23, 126, 113, 43, 113, 23, 111, 111,
    /* 120: */  58, 57, 61, 109, 23, 23, 124, 23, 57, 57,
    /* 130: */  23, 57, 61, 109, 124, 124, 57, 36, 80, 38,
    /* 140: */  8, 28, 23, 87, 41, 104, 9, 7, 124, 47,
    /* 150: */  8, 106, 43, 75, 8, 46, 96, 94, 9, 7,
    /* 160: */  123, 63, 8, 56, 11, 45, 8, 84, 103, 72,
    /* 170: */  112, 71, 115, 80, 127, 4, 52, 76, 1, 31,
    /* 180: */  117, 4, 43, 16, 17, 14, 85, 39, 4, 89,
    /* 190: */  125, 50, 9, 78, 90, 71, 4, 8, 59, 59,
    /* 200: */  53, 14, 8, 119, 57, 100, 78, 17, 48, 3,
    /* 210: */  101, 7, 9, 6, 12, 2, 119, 59, 8, 8,
    /* 220: */  119, 127, 23, 77, 67, 68, 86, 43, 23, 13,
    /* 230: */  43, 98, 111, 23, 126, 113, 43, 111, 58, 57,
    /* 240: */  61, 109, 23, 124, 109, 0};
/* prodx array */
 int prodx[] = {
    /* 0  : */  0, 1, 4, 8, 13, 16, 19, 22, 25, 28,
    /* 10 : */  33, 36, 42, 46, 50, 56, 59, 62, 65, 68,
    /* 20 : */  71, 74, 77, 82, 87, 92, 95, 98, 101, 107,
    /* 30 : */  112, 115, 120, 124, 128, 133, 136, 139, 142, 147,
    /* 40 : */  152, 157, 163, 168, 172, 175, 178, 182, 186, 190,
    /* 50 : */  194, 197, 200, 203, 206, 210, 213, 218, 221, 224,
    /* 60 : */  229, 234, 239, 244, 249, 257, 260, 263, 267, 271,
    /* 70 : */  274, 277, 280, 283, 286, 289, 293, 298, 302, 307,
    /* 80 : */  312, 317, 323, 328, 333, 338, 343, 348, 353, 358,
```

```
    /* 90 : */  363, 368, 373, 378, 383, 388, 393, 399, 401, 403,
    /* 100: */  407, 409, 411, 413, 409, 417, 409, 419, 422, 425,
    /* 110: */  428, 1, 172, 431, 433, 417, 436, 98, 428, 221,
    /* 120: */  440, 443, 446, 449, 1, 172, 452, 172, 455, 458,
    /* 130: */  172, 461, 465, 470, 475, 480, 485};
/* prods array */
 short int prods[] = {
    /* 0  : */  0, 80, 23, 0, 69, 69, 68, 0, 80, 80,
    /* 10 : */  8, 23, 0, 124, 35, 0, 124, 27, 0, 124,
    /* 20 : */  25, 0, 124, 24, 0, 122, 23, 0, 86, 38,
    /* 30 : */  88, 17, 0, 106, 23, 0, 77, 67, 86, 104,
    /* 40 : */  113, 0, 70, 9, 122, 0, 70, 7, 122, 0,
    /* 50 : */  68, 80, 28, 70, 17, 0, 110, 34, 0, 110,
    /* 60 : */  33, 0, 110, 32, 0, 110, 37, 0, 110, 40,
    /* 70 : */  0, 110, 23, 0, 75, 120, 0, 87, 23, 47,
    /* 80 : */  49, 0, 88, 88, 8, 87, 0, 104, 41, 105,
    /* 90 : */  17, 0, 94, 97, 0, 123, 24, 0, 123, 23,
    /* 100: */  0, 87, 23, 47, 75, 49, 0, 105, 105, 8,
    /* 110: */  106, 0, 56, 15, 0, 113, 43, 94, 45, 0,
    /* 120: */  107, 9, 123, 0, 107, 7, 123, 0, 75, 75,
    /* 130: */  8, 120, 0, 103, 6, 0, 71, 47, 0, 91,
    /* 140: */  42, 0, 97, 96, 56, 65, 0, 94, 94, 13,
    /* 150: */  97, 0, 63, 107, 11, 107, 0, 53, 43, 54,
    /* 160: */  45, 110, 0, 54, 54, 8, 63, 0, 102, 103,
    /* 170: */  51, 0, 126, 23, 0, 117, 111, 0, 115, 71,
    /* 180: */  49, 0, 71, 23, 47, 0, 111, 126, 74, 0,
    /* 190: */  125, 125, 127, 0, 85, 99, 0, 85, 123, 0,
    /* 200: */  78, 81, 0, 58, 92, 0, 76, 52, 79, 0,
    /* 210: */  90, 44, 0, 51, 43, 76, 45, 0, 73, 114,
    /* 220: */  0, 95, 111, 0, 93, 72, 1, 73, 0, 82,
    /* 230: */  112, 31, 116, 0, 115, 71, 117, 49, 0, 55,
    /* 240: */  116, 16, 73, 0, 66, 66, 17, 64, 0, 96,
    /* 250: */  46, 23, 84, 4, 83, 5, 0, 119, 10, 0,
    /* 260: */  100, 99, 0, 78, 125, 81, 0, 62, 50, 60,
    /* 270: */  0, 101, 20, 0, 101, 28, 0, 101, 30, 0,
    /* 280: */  101, 19, 0, 101, 29, 0, 101, 18, 0, 121,
    /* 290: */  9, 95, 0, 76, 76, 90, 79, 0, 114, 71,
    /* 300: */  49, 0, 117, 117, 8, 111, 0, 74, 4, 59,
    /* 310: */  5, 0, 74, 43, 59, 45, 0, 127, 80, 14,
    /* 320: */  120, 17, 0, 99, 23, 14, 63, 0, 83, 83,
    /* 330: */  8, 85, 0, 79, 39, 119, 65, 0, 95, 4,
    /* 340: */  57, 5, 0, 89, 4, 100, 5, 0, 78, 58,
    /* 350: */  17, 81, 0, 58, 58, 17, 92, 0, 61, 61,
    /* 360: */  3, 62, 0, 118, 118, 6, 121, 0, 118, 118,
    /* 370: */  12, 121, 0, 118, 118, 2, 121, 0, 79, 78,
    /* 380: */  119, 65, 0, 114, 71, 59, 49, 0, 100, 100,
    /* 390: */  8, 99, 0, 79, 89, 78, 119, 65, 0, 67,
    /* 400: */  0, 86, 0, 67, 36, 69, 0, 104, 0, 96,
    /* 410: */  0, 84, 0, 96, 46, 23, 0, 52, 0, 112,
    /* 420: */  98, 0, 72, 98, 0, 116, 111, 0, 98, 111,
    /* 430: */  0, 74, 0, 65, 66, 0, 65, 125, 66, 0,
    /* 440: */  78, 58, 0, 92, 57, 0, 57, 61, 0, 60,
    /* 450: */  109, 0, 109, 118, 0, 73, 57, 0, 59, 57,
    /* 460: */  0, 92, 125, 57, 0, 57, 57, 48, 61, 0,
    /* 470: */  60, 109, 101, 109, 0, 109, 109, 7, 118, 0,
    /* 480: */  109, 109, 9, 118, 0, 59, 59, 8, 57, 0};
```

```
/* These guys are for printing tokens in parser stack dumps */

  /* token table. */
   char *tokchar[] = {
        "",  /* dummy zeroth element */
        "!", "%", "&", "(", ")", "*", "+", ",", "-", "--", "..", "/",
        "//", ":", "::", ":=", ";", "<", "<=", "<>", "<empty>", "<eol>",
        "<identifier>", "<integer>", "<real>", "<stop>", "<string>",
        "=", ">", ">=", "?", "BOOLEAN", "BYTE", "CHAR", "CHARCONS",
        "CONST", "INTEGER", "MESSAGE", "OTHERWISE", "REAL", "SIGNAL",
        "SKIP", "[", "[]", "]", "_", "{", "|", "}", "~", "AltComm",
        "AltStart", "ArrayType", "Arraydims", "AssignComm", "Block",
        "BoolExp", "BoolGuard", "BoolList", "BoolPri", "BoolTerm",
        "BoolUnary", "CRange", "Command", "CommandList", "Commands",
        "ConstDecl", "ConstItem", "ConstList", "Constant", "Constructor",
        "Destination", "Expression", "Extension", "FieldList", "GCommand",
        "Goal", "Guard", "GuardComm", "IdentList", "InpGuard", "InputComm",
        "Labelsubs", "Labhead", "Labsub", "MessageDecl", "MessageItem",
        "MessageList", "MultiGrd", "Nextg", "NullComm", "OneBool",
        "OutputComm", "ParCommand", "Primary", "ProcLabel", "Process",
        "ProcessName", "Range", "RangeList", "Relop", "RepComm", "Repeat",
        "SigDecl", "SigList", "SigTyp", "SignIntCons", "SimpCommand",
        "SimpExp", "SimpleType", "Simpvar", "Source", "StructCommand",
        "StructExp", "StructTarget", "TargetVar", "TargetVars", "Term",
        "Then", "Type", "Unary", "UnsigCons", "UnsigInt", "UnsigValu",
        "VarDeclarations", "Varib", "Varitem" };

/* Lexical and token data */
char line[LINELEN+1];  /* source line */
int lx;                /* index of next character in LINE */
int errpos = 1;        /* current token index in LINE */
int oldpos = 0;        /* position of last error reported */
int prompt_len;        /* length of prompt string */
int ch;                /* next character from input file */
int token;             /* next token in input list */
semrecp lsemp;         /* current semantics assoc. with token */
semrecp appsemp;       /* semantics assoc. formed by APPLY proc. */
int tokenx;            /* index into TOKARY, LSEMPARY */
int tokary[2];         /* token queue */
semrecp lsempary[2];
char errsym[ERRSYMLEN]; /* special symbol reserved for errors */

/* Symbol table data */
symtabp symtab[HLIMIT+1];
symtabp intptr, realptr, sigptr,
        boolptr, charptr;     /* pointers to standard types */
char strtab[STRTABLEN+1];     /* string table */
int strtabx = 0;              /* index into string table */
int varadr = FIRSTADR;        /* first offset for variables in current
                                 process */
symtabp cur_proc = NULL;      /*  process currently compiled */
int plevel = GLOBLEV;         /* compile-time level; 0 = outer block */
int rlevel = GLOBLEV;         /* runtime level; 0 = global vars */
int dumlabx = 0;              /* used to define new labels for jump nodes */
int msigadr = NOSTRUCT+1;     /* to uniquely identify constructors --
                                 must start at 1, because 0 is used in I/O
                                 formats if no constructor present */
```

```
int nxtproc = FIRSTPROC;       /* first process number allocatable;
                                  inp = 0; out = 1 */
int gencode = 0;               /* generate code only if zero */

FILE *sfile, *rfile, *cfile; /* source, report and code files */
char sfilename[81], rfilename[81],
     cfilename[81];            /* source, report and code filenames */
boolean interleave;            /* allows source to be printed in code file */

int errors = 0;
int debug = 0;                 /* >0 turns on some tracing */

char *symtypename[] =
    { "reserved", "symerr", "user", "variable", "constant", "integer",
      "real", "boolean", "type", "process", "array", "message",
      "subrange", "signal", "char", "unknown" };

char *semtypename[] =
    { "none", "ident",  "strng", "type", "idlist", "arrayext",
      "fixed", "float", "add", "sub", "multiply", "divide", "mod",
      "minus", "boolean", "and", "or", "not", "equal", "gtreql", "gtr",
      "leseql", "less", "nequal", "blist", "jump", "repeat", "messnode",
      "signode","messext", "procext", "error" };
```

```
/* External variable declarations and function definitions for the CSP-i
   parser */

extern char *symtypename[ ], *semtypename[ ];
extern int statex[ ], map[ ], stk_state[ ], stk_tostate[ ], toknum[ ], tostate[ ];
extern short int popno[ ], prods[ ];
extern int insym[ ], prodx[ ], TOKWORD[ ], stack[ ], stackx;
extern char WORDS[ ], *tokchar[ ];
extern semrecp semstack[ ];

/* Lexical and token data */
extern char line[ ];             /* source line */
extern int lx;                   /* index of next character in LINE */
extern int errpos;               /* current token index in LINE */
extern int oldpos;               /* position of last error reported */
extern int prompt_len;           /* length of prompt string */
extern int ch;                   /* next character from input file */
extern int token;                /* next token in input list */
extern semrecp lsemp;            /* current semantics assoc. with token */
extern semrecp appsemp;          /* formed by APPLY procedure */
extern int tokenx;               /* index into TOKARY, LSEMPARY */
extern int tokary[ ] ;           /* token queue */
extern semrecp lsempary[ ];
extern char errsym[ ];           /* special symbol reserved for errors */

/* Symbol table data */
extern symtabp symtab[ ];
extern char strtab[ ];
extern symtabp intptr, realptr, charptr,
               boolptr;        /* pointers to standard types */
extern int strtabx;            /* index into string table */
extern int varadr ;            /* greatest offset of variables in
                                  current process */
extern symtabp cur_proc;       /* process currently being compiled */
extern int plevel;             /* compile-time level; 0 = outer block */
extern int rlevel;             /* runtime level; 0 = global vars */
extern int dumlabx;            /* used as label index for jump nodes */
extern int msigadr;            /* unique number for constructors */
extern int nxtproc;            /* next process number allocatable */
extern int gencode;            /* whether to emit code */
extern FILE *sfile, *rfile, *cfile;    /* source, report and code files */
extern char sfilename[ ], rfilename[ ],
            cfilename[ ];              /* source, report and code filenames */
extern boolean interleave;             /* include source text in code file */
extern int errors;
extern int debug;               /* >0 turns on some tracing */

/* General Utilities */
extern char *getmem( );        /* allocates heap space */
extern int strlen( );          /* returns length of a string */
extern int strcmp( );          /* is s1 == s2 ? (0, +, -) */
extern char *strcpy( );        /* copies s2 to s1 */
extern char *strcat( );        /* catenates s2 to end of s1 */
extern FILE *fopen( );         /* returns file pointer */
extern fclose( );              /* closes a file */
extern fprintf( ), fputc( ); /* output routines */
extern fscanf( ), gets( );
extern char *fgets( );         /* input routines  */
```

```
/* Functions found in CSPREP.C */
extern char upshift(), resp();
extern boolean yesresp();
extern more(), abort(), error(), symerror(), warning();
extern symtabp newsym();
extern semrecp newsem();

/* Functions found in CSPDBUG.C */
extern int dumpx();

/* Functions found in CSPSYMS.C */
extern int hashit(), keyword();
extern clearsym();
extern symtabp findsym(), makesym(), forcesym();
```

```
 /* Definitions used by both the CSP-i parser and interpreter */

/* Definitions used by wrtype() and gen_rw() */
#define ITG 0
#define REL 1
#define ARR 2
#define SGN 3
#define CHR 4
#define MSS 5

/* Intermediate code definitions */
 /* Arithmetic and logical ops */
#define ADI 1
#define ADR 2
#define SBI 3
#define SBR 4
#define MLI 5
#define MLR 6
#define DVI 7
#define DVR 8
#define MOD 9
#define NOT 10
#define NGI 11
#define NGR 12
#define EQI 13
#define EQR 14
#define GTI 15
#define GTR 16
#define LSI 17
#define LSR 18
#define AND 19
#define ORR 20

 /* Conversion operators */
#define FLO 21
#define FIX 22

 /* Input and Output */
#define MSI 23
#define MSO 24

 /* Guard delimiters */
#define GDB 25
#define GDE 26
#define EDF 27    /* end of default guard */

 /* Jump codes */
#define ALN 28    /* alternate stmt */
#define AJP 29
#define AJD 30    /* jump table entry for otherwise guard */
#define FJP 31
#define UJP 32

 /* Store operators */
#define STI 33
#define STR 34
#define STA 35
#define SOI 36    /* NB. the three SOx codes must follow directly */
```

```
#define SOR 37   /* after the three STx codes */
#define SOA 38

#define STLD 3

 /* Load operators */
#define LDI 39
#define LDR 40
#define LDA 41
#define LOI 42
#define LOR 43
#define LOA 44
#define LDS 45   /* string */
#define LSG 46   /* signal */
#define LCH 47
#define LID 48   /* load address */
#define IND 49   /* check index */
#define PSI 50   /* push integer */
#define PSR 51

 /* Miscellaneous */
#define LOC 52
#define DEF 53
#define LAB 54   /* load process labels */
#define INC 55
#define DEC 56
#define BEG 57
#define EXT 58   /* end of a process */
#define HLT 59
#define JBL 60   /* start of jump table */
#define EBL 61   /* end of jump table */
#define CRG 62   /* change guard ring */
#define NOP 63
#define STK 64
#define DMP 65

#define LASTCODE 65
```

```
 /* CSP-SKELSYMS:  Symbol table handling for skeleton files. */
 /* Copyright (C) 1984 by QCAD Systems Inc., All Rights Reserved. */
 /* Modifications by K.L. Wrench for use with the CSP-i parser. */

#include "CSPDEFS.C"

extern int strcmp(), strcpy();
extern symtabp newsym(), symtab[];
extern int rlevel;
extern rlsmem();

 /* Structures for reserved keywords */
 char *WORDS[] =
   { "BOOLEAN","BYTE","CHAR","CONST","INTEGER",
     "MESSAGE","OTHERWISE","REAL","SIGNAL","SKIP" };

 int TOKWORD[] =
   { BOOLEAN, BYTE, CHAR, CONST, INTEGER,
     MESSAGE, OTHERWISE, REAL, SIGNAL, SKIP };

/* ---------------------------------------------------------------- */

 int keyword(str)
  char *str;
   /* Determine whether an identifier is a reserved word */

  { int i = 0,
        k,res,
        j = NORESERVED - 1;

    do { /* binary search */
         k = (i+j) / 2;
         res = strcmp(str,WORDS[k]);
         if (res <= 0) j = k-1;
         if (res >= 0) i = k+1;
       }
     while (i <= j);

    if (i-1 > j) /* found reserved word */
         return(TOKWORD[k]);
      else return(IDENT_TOKX);
  } /* keyword */

/* ---------------------------------------------------------------- */

 int hashit(fsym)
  char fsym[];

  { char template[MAXTOKLEN+1];

    strcpy(template, "    ");  /* need to establish 4 chars */
    strcpy(template, fsym);
    return(( 128*(template[0] + template[2])
           + template[1] + template [3]) % HASHSIZE);
  }  /* hashit */
```

```
/* ------------------------------------------------------------------ */

 symtabp findsym(fsym)
  char fsym[];
   /* Finds a symbol, returning NULL if not there.  Used for
      variable references: error if NULL is returned. */

  { symtabp sp;

    for (sp = symtab[hashit(fsym)]; sp != NULL; sp=sp->next)
      if (strcmp(sp->sym, fsym) == 0) return(sp);
    return(NULL);
  }  /* findsym */

/* ------------------------------------------------------------------ */

 extern symtabp makesym(fsym, syt, lev)
  char fsym[];
  symtype syt;
  int lev;
   /* This returns a symbol entry if there; makes a new one if not */

  { symtabp sp;
    int hx;

    sp = findsym(fsym);
    if (sp == NULL)
      { sp = newsym(); /* need a new one if here */
        /* put at the head of the hash list*/
        strcpy(sp->sym, fsym);
        sp->symt = syt;
        hx = hashit(fsym);
        sp->next = symtab[hx];
        symtab[hx] = sp;
        sp->level = lev;
        sp->rlevel = rlevel;
      }
    return(sp);
  }  /* makesym */

/* ------------------------------------------------------------------ */

 symtabp forcesym(fsym, syt, lev)
  char fsym[];
  symtype syt;
  int lev;
   /* This forces a new symbol entry.  Use this for declarations
      to cover a previous declaration with the same name */

  { symtabp sp;
    int hx;

    hx = hashit(fsym);
    sp = newsym();
    /* put at the head of the hash list */
    strcpy(sp->sym, fsym);
    sp->symt = syt;
    sp->next = symtab[hx];
```

```
    symtab[hx] = sp;
    sp->level = lev;
    sp->rlevel = rlevel;
    return(sp);
  }  /* forcesym */

/* ---------------------------------------------------------------------- */

 clearsym(clevel,all)
  int clevel;
  boolean all;
   /* Sets the symbol table pointers to remove references to
      everything at level >= CLEVEL, assuming that level numbers
      are monotonic.  Prepares for a RELEASE of memory. */

  { symtabp sp, keep, rel, lastkeep;
    int hx, release();

    /* Don't clear the reserved words -- check, just in case */
    if (clevel<0) clevel = 0;
    for (hx = 0; hx <= HLIMIT; hx++)
     { sp = symtab[hx];  lastkeep = keep = NULL;
       while (sp != NULL)
        { if (sp->level >= clevel)  /* remove node */
            { rel = sp; sp = sp->next;
              if (lastkeep)  /* bypass node to be removed */
                   lastkeep->next = sp;
              release(rel,clevel,all);
             }

          else  /* retain node */
            { if (!keep) keep = sp;  /* mark first of nodes to be retained */
              lastkeep = sp; sp = sp->next; }
         }
      symtab[hx] = keep;
    }
  }  /* clearsym */

/* ---------------------------------------------------------------------- */

 static release(ptr, lev, all)
  symtabp ptr;
  int lev;
  boolean all;
   /* Releases the memory occupied by nodes of the symbol table */

  { int clearnode();

    if (!ptr || ptr->level < lev ) return;  /* stop releasing this chain */
    switch (ptr->symt)
     {  case VARI:
             if (! (all || ptr->usym.vsymt.canchange)) return;
              /* do not release process label nodes */
             release(ptr->usym.vsymt.vtypep, lev, all); /* type nodes */
             release(ptr->usym.vsymt.nxtvar, lev, all); /* nxtvar nodes */
             clearnode(ptr); break;
        case USER_TYPE: release(ptr->usym.utypep, lev, all);
                        clearnode(ptr); break;
```

```
        case REAL_TYPE:
        case INT_TYPE:
        case CHAR_TYPE:
        case BOOL_TYPE:
        case ARRAY_TYPE:
        case MESS_TYPE:
        case SUBRANGE:
        case SIG_TYPE:  release(ptr->usym.asymt.subtype, lev, all);
                        clearnode(ptr); break;
        case PROC_TYPE: release(ptr->usym.psymt.labptr, lev, all);
                        release(ptr->usym.psymt.nxtproc, lev, all);
                        clearnode(ptr); break;
        default:  clearnode(ptr); break;
    }

  } /* release */

/* ------------------------------------------------------------------------- */

 static clearnode(sptr)
  symtabp sptr;
   /* Release memory requirements of one node */

  { if (rlsmem( (char *) sptr, sizeof (struct symtabtype)) < 0)
      error ("Incomplete release of memory");
  }  /* clearnode */

/* ------------------------------------------------------------------------- */
```

```
 /* CSPNUM:  Number scanning for skeleton files. */
 /* Copyright (C) 1984 by QCAD Systems Inc., All Rights Reserved. */
 /* Modifications by K.L. Wrench for use with the CSP-i parser.  */

#include "CSPDEFS.C"
#include "CSPALL.C"

#define ODD(n)  ((n % 2) == 1)

 float pwr10_2[] =
   { 1.0E1, 1.0E2, 1.0E4, 1.0E8, 1.0E16, 1.0E32 }; /* Binary powers of ten */

/* ---------------------------------------------------------------------- */

 float pwr10(n)
  int n;

  { float p10;
    boolean sign;
    int px;

    sign = (n < 0);
    if (n < 0) n = -n;
    p10 = 1.0;
    if (n >= 38) { error("exponent too large -- 37 assumed");
                   p10 = 1E37; }
      else {  px = 0;
              while (n>0)
                { if ODD(n) p10 = p10 * pwr10_2[px];
                  n = n / 2; px++;
                }
            }
    return((float) (sign ? (1.0 / p10) : p10));
  }   /* pwr10 */

/* ---------------------------------------------------------------------- */

 static int get_integer()
  /* Interpret a non-null sequence of digits as an integer */

  { int v = 0;

      while ((ch>='0') && (ch<='9'))
        { v = 10*v + ch - '0';
          nextch(); }
      return(v);
   }  /* get_integer */

/* ---------------------------------------------------------------------- */

 static float get_fraction()

  { float v = 0, p = 0.1;

     while ((ch>='0') && (ch<='9'))
       { v = v+p*(float)(ch-'0');
         p = p/10.0;
         nextch();   }
```

```
    return(v);
  }  /* get_fraction */

/* ---------------------------------------------------------------------------- */

 static get_exp(rv)
  float *rv;
   /* Get the exponent part of a floating point number */

  { boolean expsign;
    int exp;

      nextch();  /* get over e or E */
      expsign = FALSE;
      if (ch == '+') nextch();
        else if (ch == '-')
               { expsign = TRUE; nextch(); }
      if ((ch>='0') && (ch<='9'))
        { exp = get_integer();
          *rv = expsign ? (*rv)/pwr10(exp) : (*rv)*pwr10(exp);
         }
       else error("missing digit after E in exponent");
  }  /* get_exp */

/* ---------------------------------------------------------------------------- */

 static finish_real(rv)
  float rv;
   /* Return a real number as the result of the lexical scan */

   { token = REAL_TOKX;
     lsemp->semt = FLT;
     lsemp->svtype = realptr;
     lsemp->usem.rval = rv;
   }   /* finish_real */

/* ---------------------------------------------------------------------------- */

 get_number()
  /* Accepts an integer, decimal or real number */

  { int v1, v2;
    float rv;

    lsemp = newsem();
    v1 = get_integer();
    if (ch == '.')
     { /* real number */
       nextch();
       rv = v1 + get_fraction();
       if ((ch == 'e') || (ch == 'E')) get_exp(&rv);
       finish_real(rv);
      }
     else if ((ch == 'e') || (ch == 'E'))
      /* integer followed by exponent part */
       { rv = v1;
         get_exp(&rv);
         finish_real(rv); }
```

```
      else
        { token = INT_TOKX;
          lsemp->semt = FIXED;
          lsemp->svtype = intptr;
          lsemp->usem.numval = vl;
        }
  } /* get_number */

/* ---------------------------------------------------------------------- */

 get_char_const()
  /* Parse a single character enclosed in single quotes */

  { nextch();
    lsemp = newsem();
    token = CHARCONS;        /* char constant token number */
    lsemp->semt = FIXED;
    lsemp->usem.numval = ch;
    lsemp->svtype = charptr;
    nextch();
    if (ch != CHAR_QUOTE) error("Illegal character constant");
      else nextch();
  } /* get_char_const */

/* ---------------------------------------------------------------------- */
```

```
 /* General functions to allocate dynamic storage and report errors */

#include "CSPDEFS.C"

extern int gencode, oldpos, errpos, errors, prompt_len, strlen();
extern FILE *rfile;
extern char *getmem();

/* ------------------------------------------------------------------ */

 semrecp newsem()
  /* Allocate storage from the heap for entries in the semantics stack */

  { char *new;
    if ((new = getmem (sizeof (struct semrec))) == NULL)
               abort("Insufficient memory! ");
     else return ((semrecp) new);
  }  /* newsem */

/* ------------------------------------------------------------------ */

 symtabp newsym()
  /* Allocate storage from the heap for symbol table entries */

  { char *new;
    if ( (new = getmem (sizeof (struct symtabtype))) == NULL)
             abort ("Insufficient memory! ");
      else return ((symtabp) new);
  }  /* newsym */

/* ------------------------------------------------------------------ */

 char upshift(c)
  /* Convert lower case characters to uppercase */

   {   (islower(c)) ? (c+'A'-'a') : c;   }

/* ------------------------------------------------------------------ */

 char resp(msg)
  char msg[];
   /* Print a message and return a single character response */

  { char ch, chstring[80];

    fprintf(stderr, "%s", msg);
    ch = (gets(chstring)) ? chstring[0] : '\n';
    putc('\n', stderr);
    return(ch);
  }  /* resp */

/* ------------------------------------------------------------------ */

 boolean yesresp (msg)
  char msg[];
   /* Query with a Y or N reply */

  { char ch;
```

```
    ch = resp(msg);
    return((ch == 'y') || (ch == 'Y'));
  }  /* yesresp */

/* ---------------------------------------------------------------------- */

 more(msg)
  char msg[];
   /* Print the string, and let the user type any character to proceed. */

  { char foo = resp(msg);    }

/* ---------------------------------------------------------------------- */

 static report_err(str,sym,msg)
  char msg[],str[],sym[];

  { register int count = errpos+prompt_len;

      while (--count > 0) fputc(' ', rfile);
      fprintf(rfile, "^\n");  /* mark error point */
      if (strlen(sym)) fprintf(rfile, "%s: %s %s\n", str,sym,msg);
            else fprintf(rfile, "%s: %s\n", str, msg);
  }  /* report_err */

/* ---------------------------------------------------------------------- */

 abort(msg)
  char msg[];

  { report_err("ABORT ","", msg); exit(1);  }

/* ---------------------------------------------------------------------- */

 error(msg)
  char msg[];

  { gencode = -1;    /* stop generating code */
    if (oldpos != errpos)
     { report_err("ERROR ","",msg);
       if (errors>MAXERRORS) abort("Error limit exceeded");
       errors++;  oldpos = errpos;
       more("Type any character to continue: ");
     }
   }  /* error */

/* ---------------------------------------------------------------------- */

 warning(msg)
  char msg[];

  { report_err("WARNING ","",msg); }
```

```
/* ----------------------------------------------------------------------- */

 symerror(sym, msg)
  char sym[], msg[];

  { gencode = -1;    /* stop generating code */
    if (oldpos != errpos)
     { report_err("SYMERROR ",sym,msg);
       if (errors>MAXERRORS) abort("Error limit exceeded");
       errors++;  oldpos = errpos;
       more("Type any character to continue: ");
     }
  }  /* symerror */

/* ----------------------------------------------------------------------- */
```

```
/* Functions for error_recovery in the CSP-i parser */

#include "CSPDEFS.C"
#include "CSPALL.C"

/* ------------------------------------------------------------------------ */

 copy_stack(mstack, mstackx, mcstate, stack, stackx, cstate, jstx)
  int mstack[ ], mstackx, mcstate, stack[ ], *stackx, *cstate, jstx;

  { register int stx;

    if ((jstx<0) || (jstx>mstackx)) abort("ERROR RECOVERY BUG");
    for (stx = 0; stx <= jstx; stx++)
      stack[stx] = mstack[stx];
    *stackx = jstx;
    *cstate = (jstx == mstackx) ?  mcstate : mstack[jstx+1];
  }  /* copy_stack */

/* ------------------------------------------------------------------------ */

 err_pushread(cstate, stack, stackx)
  int cstate, stack[ ], *stackx;
   /* Do the push part of a READSTATE */

  { (*stackx)++;
    if (*stackx>STACKSIZE) abort("stack overflow");
    stack[*stackx] = cstate;
  }  /* err_pushread */

/* ------------------------------------------------------------------------ */

 boolean trial_parse(cstate, stack, stackx)
  int *cstate, stack[ ], stackx;
   /* Parses from current read state through the inserted and the
      error token; if successful, returns TRUE */

  { int rx;
    boolean trial = TRUE;  /* until proven otherwise */

    while (*cstate!=0)
      { if (*cstate < READSTATE)
          { /* a reduce state */
            if (debug > 3) stk_dump("E*Reduce", stack, stackx, *cstate);
            if (popno[*cstate] == 0)
              { /* empty production */
                err_pushread(stk_state[statex[*cstate]], stack, &stackx);
                *cstate = stk_tostate[statex[*cstate]];
               }
             else
              { /* non-empty production */
                stackx = stackx - popno[*cstate] + 1;
                rx = statex[*cstate];    /* compute the GOTO state */
                *cstate = stack[stackx];
                while ((stk_state[rx]!=*cstate) && (stk_state[rx]!=0)) rx++;
                *cstate = stk_tostate[rx];
              }
          }
```

```
        else if (*cstate < LOOKSTATE)
          { /* a read state */
            next_token();  /* need a token now */
            if (debug > 3) stk_dump("E*Read", stack, stackx, *cstate);
            rx = statex[*cstate];
            while ((toknum[rx]!=0) && (toknum[rx]!=token)) rx++;
            if (toknum[rx] == 0)      /* failure */
                return(FALSE);
              else
              { /* did read something */
                err_pushread(*cstate, stack, &stackx);
                *cstate = tostate[rx];
                tokenread();  /* scan the token */
                if (tokenx>1) return(trial);   /* successful */
               }
          }
          else
            { /* lookahead state */
              next_token();  /* need a token now */
              if (debug > 3) stk_dump("E*Look", stack, stackx, *cstate);
              rx = statex[*cstate];
              while ((toknum[rx]!=0) && (toknum[rx]!=token)) rx++;
              *cstate = tostate[rx];
             }
      }
    return(trial);
  }  /* trial_parse */

/* ---------------------------------------------------------------------- */

 incr_errsym()
  /* Note that this procedure assumes ASCII */

  { if (errsym[ERRSYMLEN-2] == 'Z')
      { errsym[ERRSYMLEN-3]++;          /* incrementing a char */
        errsym[ERRSYMLEN-2] = 'A';
      }
     else errsym[ERRSYMLEN-2]++;
   }  /* incr_errsym */

/* ---------------------------------------------------------------------- */

 make_default(tokx)
  int tokx;
   /* Creates a default token data structure */

   { semrecp semp;
     char sym[MAXTOKLEN];

      switch (tokx)
       { case CHARCONS:
         case INT_TOKX:
              semp = newsem();
              semp->semt = FIXED;
              semp->usem.numval = 1;
              break;
         case REAL_TOKX:
              semp =  newsem();
```

```
            semp->semt = FLT;
            semp->usem.rval = 1.0;
            break;
      case IDENT_TOKX:
            semp =  newsem();
            semp->semt = IDENT;
            semp->usem.idsemt.symp = makesym(errsym, SYMERR, 0);
            incr_errsym();
            break;
      case STR_TOKX:
            semp =  newsem();
            semp->semt = STRNG;
            semp->usem.stx = 0;  /* default string at origin */
            break;
      default:
            semp = NULL;
            break;
      } /* switch tokx */
   lsemary[0] = semp;
  }  /* make_default */

/* ------------------------------------------------------------------------ */

int error_recovery(mstack, mstackx, mcstate)
 int mstack[], *mstackx, mcstate;

 { int stack[STACKSIZE+1],  /* local copy of stack */
       stackx,              /* local stack pointer */
       cstate,              /* local state */
       jstx,                /* temporary stack limit */
       rx, tl;              /* index into TOKNUM table */

   if (debug > 3) fprintf(rfile, "Going into ERROR RECOVERY\n");
   while (TRUE)
    { jstx = *mstackx;
      while (jstx>=0)
        { copy_stack(mstack, *mstackx, mcstate, stack, &stackx,
                     &cstate, jstx);
          rx = statex[cstate];
          while (toknum[rx]!=0)
            { /* scan through legal next tokens */
              if (debug > 3) fprintf(rfile, "...starting trial parse\n");
              tokary[0] = toknum[rx];  /* the insertion */
              tokenx = 0;
              if (trial_parse(&cstate, stack, stackx))
                   goto got_it;  /* it clicked! */
              rx++;
              if (toknum[rx]!=0)
                 copy_stack(mstack, *mstackx, mcstate, stack, &stackx,
                            &cstate, jstx);
            }
          jstx--;  /* reduce stack */
         }
      if (token == STOP_TOKX)
        { /* empty stack, no more tokens */
          cstate = 0;  /* halt state */
          tokenx = 2;
          jstx = 0;  /* bottom of stack */
```

```
          goto finish;
        }
      if (debug > 3)
        { fprintf(rfile, "...dropping token ");
          tl = wrtok(tokary[1]);
          fputc('\n', rfile);
        }
      tokenx = 2;
      next_token();
      if (debug > 3)
        { fprintf(rfile, "New token ");
          tl = wrtok(token);
          fputc('\n', rfile);
        }
    }
  got_it:  /* found a solution */
    copy_stack(mstack, *mstackx, mcstate, stack, &stackx, &cstate, jstx);
    if (debug > 3)
      { fprintf(rfile, "insertion of ");
        tl = wrtok(tokary[0]);
        fprintf(rfile, " succeeded\n");
      }
    make_default(tokary[0]);
    tokenx = 0;  /* forces a 'real' rescan of the insertion */
    if (jstx<*mstackx) cstate = stack[jstx+1];
      else cstate = mcstate;  /* cstate returned */
  finish:
    *mstackx = jstx;
    if (debug > 3) fprintf(rfile, "Ending error recovery\n");
    return(cstate);
  }  /* error_recovery */

/* ----------------------------------------------------------------------- */
```

```
/* Functions for the LEXICAL ANALYZER */

#include "CSPDEFS.C"
#include "CSPALL.C"

extern float pwr10();

/* ---------------------------------------------------------------------- */

static grabline()
  /* Used by getline */

  { char *l = line;
    char *eof_condition;
    register int len;

    /* Note: using stdin instead of sfile below allows
       the backspace key to work on input lines as you type. */

      eof_condition = fgets(line, LINELEN, sfile);
      len = strlen(line);
      if (eof_condition == NULL)
        { *(l+len) = EOF; *(++l+len) = EOS; }
       else
         { fprintf(rfile, "%s", line);
           if (interleave)
             if (line[len-1] != '\n') fprintf(cfile, ";  %s\n", line);
                 else fprintf(cfile, ";  %s", line);
         }
      lx = 0; oldpos = -1;
   }  /* grabline */

/* ---------------------------------------------------------------------- */

getline()
  /* Read the next source line, when nextch exhausts the current one */

  { if (strcmp(sfilename, "") == 0)
      /* prompt if from the console file */
       printf("> ");
    grabline();
  }  /* getline */

/* ---------------------------------------------------------------------- */

nextch()
  /* Gets next character from line */

  { if (lx >= strlen(line)) getline();
    ch = line[lx];
    /* don't move past an eof mark */
    if (ch != EOF) lx++;
  }  /* nextch */

/* ---------------------------------------------------------------------- */

skipblanks()
  /* This considers backslash as an open and close-comment;
```

```
    comments may run over multiple lines, but may not be nested */

  { do
     { while (ch == ' ') nextch();
       if (ch == '\\')
         { /* open a comment */
           do nextch(); while ((ch != '\\') && (ch != EOF));
           if (ch == EOF) error("unclosed comment");
             else nextch();
         }
      } while (ch == ' ');
   } /* skipblanks */

/* ------------------------------------------------------------------------ */

 static putstrch(ch)
  char ch;

  { if (strtabx > STRTABLEN)
      abort("String table overflow ... please abort");
    strtab[strtabx++] = ch;
  }  /* putstrch */

/* ------------------------------------------------------------------------ */

 putstr(str)
  char str[];
   /* Inserts a string (str) into the string table */

  { register int sx;

    for (sx = 0; sx <= strlen(str); sx++) putstrch(str[sx]);
    putstrch(EOS);
  }   /* putstr */

/* ------------------------------------------------------------------------ */

 get_symbol()

  { register int sx = 0;
    char sym[MAXTOKLEN];
    symtabp stp;

    /* keep snarfing alphanumeric characters; up to the first
       MAXTOKLEN of them will be put in the symbol spelling */

    while (isalpha(ch) || isdigit(ch) || (ch == '_'))
     { if (sx < MAXTOKLEN-1) sym[sx++] = upshift(ch);
       nextch();
      }

    sym[sx] = '\0';
    if ( (token = keyword(sym)) == IDENT_TOKX)
      {  /* not a keyword: ordinary identifier */
        stp = makesym(sym, USER, 0);  /* the default level is 0 */
        lsemp =  newsem();
        lsemp->semt = IDENT;
        lsemp->usem.idsemt.symp = stp;  }
```

```
    else { /* a reserved keyword */
           lsemp = NULL;
         }
  }  /* get_symbol */

/* -------------------------------------------------------------------- */

get_string()
 /* Scans a string, putting it into the string table, and setting
    up the semantic record for it correctly.  Removing the "&&
    (ch != EOLCH)" clause in the WHILE loop below will allow strings
    to run over the end of a line by storing embedded EOLCH's. */

 { boolean end_of_string;

   nextch();  /* get past the first quote mark */
   lsemp =  newsem();
   lsemp->semt = STRNG;
   lsemp->usem.stx = strtabx;
   lsemp->svtype = NULL;
   do {
     while ((ch != EOF) && (ch != EOLCH)
           && (ch != STRING_QUOTE))
       { putstrch(ch);
         nextch();
       }
     end_of_string = TRUE;
     /* peek ahead a bit to see if there's a doubled quote */
     if (ch == STRING_QUOTE)
      { nextch();
        if (ch == STRING_QUOTE)
          { end_of_string = FALSE;
            putstrch(ch);
            nextch();
          }
       }
      else if ((ch == EOF) || (ch == EOLCH)) error("unterminated string");
     } while (!end_of_string);
   putstrch(EOS);
   token = STR_TOKX;
 }  /* get_string */

/* -------------------------------------------------------------------- */

get_token()
 /* Pascal-style lexical analyzer -- sets TOKEN to token number */

 { lsemp = NULL;  /* default case - merely uses existing semrec */
   skipblanks();
   errpos = lx;
   if (isalpha(ch))
     { get_symbol();
       if (lsemp != NULL)
         { lsemp->usem.idsemt.vext = NULL;
           if (lsemp->usem.idsemt.symp->symt == VARI)
                lsemp->svtype = lsemp->usem.idsemt.symp->usym.vsymt.vtypep;
             else lsemp->svtype = NULL;  }
      }
```

```
else
 if (isdigit(ch)) get_number();
  else switch (ch)
  {
   case STRING_QUOTE:  get_string(); break;
   case CHAR_QUOTE:
          get_char_const(); /* token = 36 */
          break;

   case DBUGCHAR:  idebug(); nextch(); get_token(); break;
   case PRDUMP:
   case STACDUMP:
          dumps(ch); nextch(); get_token(); break;
   case '!':  nextch(); token = 1;  /* ! */
              break;
   case '%':  nextch(); token = 2;  /* % */
              break;
   case '&':  nextch(); token = 3;  /* & */
              break;
   case '(':  nextch(); token = 4;  /* ( */
              break;
   case ')':  nextch(); token = 5;  /* ) */
              break;
   case '*':  nextch(); token = 6;  /* * */
              break;
   case '+':  nextch(); token = 7;  /* + */
              break;
   case ',':  nextch(); token = 8;  /* , */
              break;
   case '-':  nextch();
              if (ch == '>')
                  { nextch(); token = 10;  /* -> */  }
                else token = 9;  /* - */
              break;
   case '.':  nextch();
              if (ch == '.')
                 { nextch(); token = 11;  /* .. */  }
               else error("Illegal character");
              break;
   case '/':  nextch();
              if (ch == '/')
                 { nextch(); token = 13;  /* // */  }
                else token = 12;  /* / */
              break;
   case ':':  nextch();
              if (ch == ':')
                 { nextch(); token = 15;  /* :: */  }
                else if (ch == '=')
                        { nextch(); token = 16;  /* := */  }
                      else token = 14;  /* : */
              break;
   case ';':  nextch(); token = 17;  /* ; */
              break;
   case '<':  nextch();
              if (ch == '=')
                 { nextch(); token = 19;  /* <= */  }
                else if (ch == '>')
                        { nextch(); token = 20;  /* <> */  }
```

```
                        else token = 18;  /* < */
                break;
      case '=':  nextch(); token = 28;  /* = */
                break;
      case '>':  nextch();
                if (ch == '=')
                   { nextch(); token = 30;  /* >= */  }
                  else token = 29;  /* > */
                break;
      case '?':  nextch(); token = 31;  /* ? */
                break;
      case '[':  nextch();
                if (ch == ']')
                   { nextch(); token = 44;  /* [] */  }
                  else token = 43;  /* [ */
                break;
      case ']':  nextch(); token = 45;  /* ] */
                break;
      case '_':  nextch(); token = 46;  /* _ */
                break;
      case '{':  nextch(); token = 47;  /* { */
                break;
      case '|':  nextch(); token = 48;  /* | */
                break;
      case '}':  nextch(); token = 49;  /* } */
                break;
      case '~':  nextch(); token = 50;  /* ~ */
                break;
      default:   if (ch == EOF)
                   { error("Incomplete source file");
                     token = STOP_TOKX; }
                  else if (ch == EOLCH)
                        { nextch();
                          /* go find another (significant) character */
                          get_token(); }
                      else
                        { error("illegal character");
                          nextch();
                          get_token();  /* try again */
                         }
                break;
    } /* end switch */
  } /* get_token */

/* ----------------------------------------------------------------------- */

 tokenread()

  { tokenx++; }

/* ----------------------------------------------------------------------- */

 next_token()

  { if (tokenx>1)
      { tokenx = 1;
        get_token();  /* goes into token, lsemp */
        tokary[1] = token;
```

```
        lsemary[1] = lsemp;
      }
    else
      { /* is in tokary */
        token = tokary[tokenx];
        lsemp = lsemary[tokenx];
      }
  } /* next_token */

/* ---------------------------------------------------------------------- */
```

```
/* Top level functions to initialise and operate the parser */

#include "CSPDEFS.C"
#include "CSPALL.C"

/* ------------------------------------------------------------------------ */

 static parse_pushread(cstate, semp)
  int cstate;
  semrecp semp;
   /* Do the push part of a READSTATE */

   { stackx++;
     if (stackx>STACKSIZE) abort("stack overflow");
     semstack[stackx] = semp;
     stack[stackx] = cstate;
   }  /* parse_pushread */

/* ------------------------------------------------------------------------ */

 parser()
  /* Carries out a complete parse, until the halt state is seen
     -- same as empty stack */

  { int cstate = START_STATE,
        rx;
    semrecp tsemp;

    stackx = -1;
    parse_pushread(STK_STATE_1, NULL);
    while (cstate != 0)
     { if (cstate < READSTATE)
          { /* a reduce state */
            if (debug > 0) stk_dump("Reduce", stack, stackx, cstate);
            tsemp = NULL;
            if (map[cstate] != 0)
              /* the semantics action */
                { apply(map[cstate], popno[cstate], tsemp);
                  tsemp = appsemp; }
            if (popno[cstate] == 0)
              { /* empty production */
                parse_pushread(stk_state[statex[cstate]], tsemp);
                cstate = stk_tostate[statex[cstate]];
               }
             else
               /* non-empty production:
                  semantics is preserved on a unit production A --> w,
                  where |w| == 1, unless something is in TSEMP.  Note that
                  if w is nonterminal, the production may be bypassed. */

              {  stackx = stackx - popno[cstate] + 1;
                 if (tsemp != NULL) /* something valid in tsemp */
                      semstack[stackx] = tsemp;
                   else if (popno[cstate] != 1)
                           semstack[stackx] = NULL;
                      /* else preserve value of semstack[stackx] */
                 /* compute the GOTO state */
                 rx = statex[cstate];
```

```
                cstate = stack[stackx];
                while ((stk_state[rx]!=cstate) && (stk_state[rx]!=0)) rx++;
                cstate = stk_tostate[rx];
              }
        }
    else if (cstate < LOOKSTATE)
      {  /* a read state */
          next_token();  /* need next token now */
          if (debug > 2) stk_dump("Read", stack, stackx, cstate);
          rx = statex[cstate];
          while ((toknum[rx]!=0) && (toknum[rx]!=token)) rx++;
          if (toknum[rx] == 0)
             {  error("syntax error");
                cstate = error_recovery(stack, &stackx, cstate);
              }
           else
             { parse_pushread(cstate, lsemp);
               cstate = tostate[rx];
               tokenread();  /* token has been scanned */
              }
        }
        else  /* lookahead state */
           { next_token();  /* need another token now */
             if (debug > 2) stk_dump("Look", stack, stackx, cstate);
             rx = statex[cstate];
             while ((toknum[rx]!=0) && (toknum[rx]!=token)) rx++;
             cstate = tostate[rx];
           }
    }
   end_sem();
 }   /* parser */

/* --------------------------------------------------------------------- */

static putsym(tsym, tv, c)
 char tsym[];
 int tv;
 symtype c;
  /* Used by init_sym_table */

  { symtabp symp;

     symp = makesym(tsym, c, -1);
     if (c == CONSVAR) { symp->usym.csymt.consival = tv;
                         symp->usym.csymt.constyp = BOOLS; }
       else if (c == PROC_TYPE)
             { symp->usym.psymt.procnum = tv;
               symp->usym.psymt.noprocs = 1;
               symp->usym.psymt.locals = symp->usym.psymt.dim = 0;
               symp->usym.psymt.labptr = symp->usym.psymt.nxtproc =
                                symp->usym.psymt.parproc = NULL;
               symp->usym.psymt.defined = TRUE;
              }
        else symp->usym.tokval=tv;
   }  /* putsym */
```

```
/* ---------------------------------------------------------------------- */

 static init_sym_table()
  /* Initialize reserved identifiers (put them in the symbol table) */

    { putsym("FALSE", 0, CONSVAR);
      putsym("TRUE", 1, CONSVAR);
      putsym("KBD", 0, PROC_TYPE);       /* Input process */
      putsym("SCREEN", 1, PROC_TYPE);    /* Output process */
    } /* init_sym_table */

/* ---------------------------------------------------------------------- */

 inittables()
  /* Parser initialization */

  { register int sx;

    strcpy(errsym, "ERR$AA");
    lsemparyl[0] = NULL;
    lsemparyl[1] = NULL;
    lsemp = NULL;
    putstr("ERROR");  /* default error string */
    tokenx = 2;       /* no token queue */

    for (sx = 0; sx <= HLIMIT; sx++)
       symtab[sx] = NULL;  /* initialize symbol table */
    for (sx = 0; sx <= STACKSIZE; sx++) semstack[sx] = NULL;

    init_sym_table();
    init_sem();
    strcpy(line, "");   /* fake a new line */
    lx = 1;             /* any non-zero number will do here */
    nextch();           /* fetch the first character, forcing a line read */
  }  /* inittables */

/* ---------------------------------------------------------------------- */

 openfiles()
  /* Opens 'source', 'listing' and intermediate 'code' files */

   { int fillen;
     boolean success;

    /* first, get the source file */
    do {
      printf("What source file? ");
      if (gets(sfilename) && (strcmp(sfilename, "") != 0))
          { fillen = strlen(sfilename);
            if ( (fillen < 4) || (sfilename[fillen-4] != '.'))
                   strcat(sfilename,".csp");
            sfile = fopen(sfilename, READ_MODE);
          }
       else sfile = fopen("con:", READ_MODE);
      prompt_len = (sfilename[0] == EOS) ? 2 : 0;
      success = (sfile != NULL);
      if (!success) printf("file doesn't exist; try again\n");
    } while (!success);
```

```
  /* now, get the report file */
  do {
    printf("What report file? ");
    if (gets(rfilename) && (strcmp(rfilename, "") != 0)) {
      rfile = fopen(rfilename, READ_MODE);
      success = (rfile == NULL);
      fclose(rfile);
      if (!success)
        success = yesresp("..already exists, purge it? ");
      }
    else { strcpy(rfilename, "con:");
           success = TRUE; }
  } while (!success);
  rfile = fopen(rfilename, WRITE_MODE);

  /* get code file */
  do {
    printf("What code file? ");
    if (gets(cfilename) && (strcmp(cfilename, "") != 0)) {
      fillen = strlen(cfilename);
      if ( (fillen < 4) || (cfilename[fillen-4] != '.'))
                 strcat(cfilename,".cod");
      cfile = fopen(cfilename, READ_MODE);
      success = (cfile == NULL);
      fclose(cfile);
      if (!success)
        success = yesresp("..already exists, purge it? ");
      }
    else { strcpy(cfilename, "con:");
           success = TRUE; }
  } while (!success);
  cfile = fopen(cfilename, WRITE_MODE);
  interleave = yesresp("Put source as comments in code file? ");
 }  /* openfiles */

/* ------------------------------------------------------------------------ */

main()
 /* Main function for CSP-i parser */

{ printf("CSP-i [an LALR(1) parser vs. 1-Nov-86]\n");
  openfiles();
  inittables();
  parser();         /* does it all */
  fclose(sfile);
  fflush(rfile);
  fclose(rfile);
  fflush(cfile);
  fclose(cfile);
}  /* main */

/* ------------------------------------------------------------------------ */
```

```
/* CSP-DBUG:  Skeleton file debugging routines. */
/* Copyright (C) 1984 by QCAD Systems Inc., All Rights Reserved. */
/* Modifications by K.L. Wrench for use with the CSP-i parser. */

#include "CSPDEFS.C"
#include "CSPALL.C"

/* ------------------------------------------------------------------------ */

 int wrtok(tx)
  int tx;
   /* Writes the print name of the TX'th token, returning the number
      of characters output */

  { fprintf(rfile, "%s", tokchar[tx]);
    return(strlen(tokchar[tx]));
  }  /* wrtok */

/* ------------------------------------------------------------------------ */

 wrprod(prx)
  int prx;
   /* Write out the PRX'th production (a series of tokens) */

  { int tl;

    prx = prodx[prx];
    tl = wrtok(prods[prx]);
    fprintf(rfile, " ->");
    while (prods[++prx] != 0)
      { fputc(' ', rfile);
        tl = wrtok(prods[prx]);
      }
  }  /* wrprod */

/* ------------------------------------------------------------------------ */

 dump_sym(indent, symp)
  int indent;
  symtabp symp;
   /* Output information on the given symbol table entry. */

  { if (symp != NULL)
      { fputc('\n', rfile);
        while (indent-- > 0) fputc(' ', rfile);
        fprintf(rfile, "%s (%-8.8s %ld %ld  ", symp->sym,
                (symp->symt <= LAST_SYMTYPE) ? symtypename[symp->symt] : "",
                symp->level, symp->rlevel);
        switch (symp->symt)
         {
           case RESERVED:
           case SYMERR: break;
           case USER:   fprintf(rfile, "undeclared"); break;
           case VARI:   fprintf(rfile, "%ld %ld", symp->usym.vsymt.saddr,
                             symp->usym.vsymt.canchange);
                        dump_sym(indent+2, symp->usym.vsymt.vtypep);
                        dump_sym(indent+4, symp->usym.vsymt.nxtvar);
                        break;
```

```
        case USER_TYPE: dump_sym( indent+2, symp->usym.utypep); break;
        case BOOL_TYPE:
        case CHAR_TYPE:
        case INT_TYPE:
        case REAL_TYPE:
              fprintf(rfile, "%ld NULL", symp->usym.asymt.size);
              break;
        case SUBRANGE:
        case ARRAY_TYPE:
              fprintf(rfile, "%3d %3d %3d ", symp->usym.asymt.size,
                      symp->usym.asymt.udim, symp->usym.asymt.ldim);
              dump_sym( indent+2, symp->usym.asymt.subtype);
              break;
        case SIG_TYPE:
        case MESS_TYPE:
              fprintf(rfile, "%3d %3d %3d", symp->usym.asymt.size,
                      symp->usym.asymt.ldim, symp->usym.asymt.udim);
              dump_sym( indent+2, symp->usym.asymt.subtype);
              break;
        case CONSVAR:
            switch (symp->usym.csymt.constyp)
                    { case FIXED: fprintf(rfile, "Integer %5d",
                           symp->usym.csymt.consival); break;
                      case FLT: fprintf(rfile, "Real %10f",
                           symp->usym.csymt.consrval); break;
                      case STRNG: fprintf(rfile, "String %s",
                           &strtab[symp->usym.csymt.consival]); break;
                      case BOOLS: (symp->usym.csymt.consival == 0) ?
                                 fprintf(rfile, "Boolean FALSE") :
                                 fprintf(rfile, "Boolean TRUE");
                               break;
                      default:  fprintf(rfile, "Not allowed"); break;
                    }
            break;
       case PROC_TYPE:
              fprintf(rfile, "%2d %2d %2d %ld ",
                      symp->usym.psymt.procnum, symp->usym.psymt.noprocs,
                      symp->usym.psymt.locals, symp->usym.psymt.dim);
              if (symp->usym.psymt.defined) fprintf(rfile,"DEFINED ");
                  else fprintf(rfile, "UNDEFINED ");
              fprintf(rfile, "%3d", (symp->usym.psymt.parproc) ?
                      symp->usym.psymt.parproc->usym.psymt.procnum : -99);
              dump_sym( indent+2, symp->usym.psymt.labptr);
              dump_sym( indent, symp->usym.psymt.nxtproc);
              break;
     default: fprintf(rfile, "????" ); break;
     }
   fprintf(rfile, ")\n" );
  }
 }  /* dump_sym */

/* ---------------------------------------------------------------------- */

dumpx (x)
 int x;
  /* Dump topmost x records of semantic stack */

 { register int i;
```

```
    fprintf(rfile, "\n%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% \n");
    fprintf(rfile, "Semantic stack -- top %d elements \n", x);
    for (i=0; i<x; i++)
       { putc('*', rfile);
         dump_sem(0, semstack[stackx-i]);
         putc('\n', rfile);
        }
    fprintf(rfile, "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% \n");
  }  /* dumpx */

/* ----------------------------------------------------------------------- */

 idlist_dump(sp)
  semrecp sp;
   /* Dump contents of an idlist semantic stack node */

  { while (sp != NULL)
       { if (sp->usem.insemt.idsymp != NULL)
              fprintf(rfile, "%-8.8s  ", sp->usem.insemt.idsymp->sym);
           else fprintf(rfile, "Sym=NULL  ");
         sp = sp->usem.insemt.idnext;
        }
  }  /* idlist_dump */

/* ----------------------------------------------------------------------- */

 blist_dump(sp,i)
  semrecp sp;
  int i;  /* indent */
   /* Dump contents of a blist semantic stack node */

   { while (sp != NULL)
        { if (sp->usem.bsemt.blexp != NULL)
                dump_sem( i+2, sp->usem.bsemt.blexp);
            else fprintf(rfile, "Blexp = null ");
          sp = sp->usem.bsemt.blnext;
        }
   }  /* blist_dump */

/* ----------------------------------------------------------------------- */

 dump_sem( indent, semstk)
  int indent;
  semrecp semstk;
   /* Output a semantic stack record */

  { if (semstk != NULL)
      { fputc('\n', rfile);
        while (indent-- > 0) fputc(' ', rfile);
        fprintf(rfile, "%-6.6s: ",
           (semstk->semt <= LAST_SEMTYPE) ? semtypename[semstk->semt] : "");
        if (semstk->svtype == NULL) fprintf(rfile, "++++ NULL  ");
          else
            fprintf(rfile, "++++ %s  ", symtypename[semstk->svtype->symt]);
        switch (semstk->semt)
          {
            case NONE:  break;
            case STRNG:  fprintf(rfile, "%s\n", &strtab[semstk->usem.stx]);
```

```
             break;
case IDENT:  dump_sym( indent+2, semstk->usem.idsemt.symp);
             if (semstk->usem.idsemt.vext != NULL)
                 { fprintf(rfile, "Vext: ");
                   dump_sem( indent+2, semstk->usem.idsemt.vext);
                 }
             break;
case FIXED:  fprintf(rfile, "%ld", semstk->usem.numval);
             dump_sym( indent+1, semstk->svtype);
             break;
case FLT:  fprintf(rfile, "%10f", semstk->usem.rval);
           dump_sym( indent+1, semstk->svtype);
           break;
case BOOLS:  if (semstk->usem.numval > 0) fprintf(rfile, "TRUE");
                else fprintf(rfile, "FALSE");
             dump_sym( indent+1, semstk->svtype);
             break;
case TYP_NODE: dump_sym( indent, semstk->usem.the_typep);
               break;
case IDLIST_NODE: idlist_dump(semstk); break;
case PROC_EXT:
case AR_EXT: while (semstk != NULL)
               { dump_sem( indent+2, semstk->usem.arsemt.ax);
                 semstk = semstk->usem.arsemt.arextnext;
                 if (semstk != NULL)
                    fprintf(rfile, "\n%s : ",
                           semtypename[semstk->semt]);
               }
              break;
case MES_EXT:
          fprintf(rfile, "Offset = %d", semstk->usem.offset);
          break;
case BLIST_NODE: blist_dump(semstk,indent); break;
case REP_NODE:
case JP_NODE:
   fprintf(rfile, "JP Labels %5d %5d ",
           semstk->usem.jsemt.jplabl, semstk->usem.jsemt.jplab2);
   break;
case ALT_NODE:
  fprintf(rfile, "Alt Labels %5d %5d %5d %5d ",
      semstk->usem.altsemt.altlabl, semstk->usem.altsemt.altlab2,
      semstk->usem.altsemt.altlab3, semstk->usem.altsemt.altlab4);
  dump_sem( indent+4,semstk->usem.altsemt.altext);
  break;
case SIG_NODE: dump_sym( indent+2, semstk->usem.sigsemt.sigvar);
               break;
case MESS_NODE:
     dump_sym( indent+2, semstk->usem.gsemt.messvar);
     if (semstk->usem.gsemt.messexp->semt == BLIST_NODE)
           blist_dump( semstk->usem.gsemt.messexp, indent);
       else idlist_dump( semstk->usem.gsemt.messexp);
     break;
case NEG_NODE:
case NOT_NODE:
     if (semstk->usem.treeun != NULL)
          fprintf(rfile, "Child type = %s",
                 semtypename[semstk->usem.treeun->semt]);
     break;
```

```
          default:
              if (semstk->semt >= LAST_SEMTYPE)
                   fprintf(rfile, " ... user form");
                else /* all binary tree nodes  */
                   { if (semstk->usem.msemt.treel != NULL)
                       fprintf(rfile, "Left type = %s ",
                              semtypename[semstk->usem.msemt.treel->semt]);
                     if (semstk->usem.msemt.treer != NULL)
                        fprintf(rfile, "Right type = %s ",
                               semtypename[semstk->usem.msemt.treer->semt]);
                   }
              break;
        }
      }
    else fprintf(rfile, "semstack = null");
   fputc('\n', rfile);
  }  /* dump_sem */

/* ---------------------------------------------------------------------- */

 stk_dump(kind, stack, stackx, cstate)
  char kind[];
  int stack[], stackx, cstate;
   /* Produce a symbolic dump of the parser stack */

  { int tl, ll;
    register int sx;
    int count;

    if (debug>2)
      { fprintf(rfile, "%s", kind);
        if (cstate >= READSTATE)
           { fprintf(rfile, ", on token ");
             tl = wrtok(token); }
        fputc('\n', rfile);  /* ", memavail %ld\n", memavail);*/
      }
    if (cstate < READSTATE)
      { /* reduce state */
        if (debug > 1)
          {  /* complete stack dump */
             if (stackx > 15)
                  { fprintf(rfile, "  ###\n");
                    ll = stackx-15; }
               else ll = 1;
             for (sx = ll; sx <= stackx; sx++)
                { /* fprintf(rfile, "  %3d ", stack[sx]); */
                  tl = wrtok(insym[(sx == stackx) ? cstate : stack[sx+1] ]);
                  count = MAXTOKLEN - tl + 1;
                  while (count-- > 0) fputc(' ', rfile);
                  dump_sem(0, semstack[sx]);
                  fputc('\n', rfile);
                 }
          }
        wrprod(cstate);
        fputc('\n', rfile);
      }
```

```
    /* don't let this roll off the top of the screen */
    idebug();
  }  /* stk_dump */

/* ------------------------------------------------------------------- */

 static show_sym()
  /* Asks for a symbol, dumps the symbol table entry for it */

    { symtabp sp;
      register int sx;
      char str[81];

      printf("What symbol? ");
      gets(str);
      for (sx = 0; str[sx] != '\0'; sx++) str[sx] = upshift(str[sx]);
      sp = findsym(str);
      if (sp != NULL) dump_sym(0, sp);
        else printf("Unknown symbol\n");
      putchar('\n');
    }  /* show_sym */

/* ------------------------------------------------------------------- */

 static dump_all()
  /* Show everything in the symbol table */

    { register int hx;
      symtabp sp;

      for (hx = 0; hx <= HLIMIT; hx++)
        for (sp = symtab[hx]; sp != NULL; sp=sp->next)
          if ((sp->symt!=RESERVED) && (sp->symt!=SYMERR))
             { /* report only the nontrivial stuff */
               fprintf(rfile, "%s ", sp->sym); }
      fputc('\n', rfile);
    }  /* dump_all */

/* ------------------------------------------------------------------- */

 static set_debug()
  /* Prompts for a debug level number */

    { char str[81];

      printf("Set debug level to (0, 1, ...)? ");
      if (gets(str) && (str[0] != '\0')) sscanf(str, "%d", &debug);
        else debug = 0;
    }  /* set_debug */

/* ------------------------------------------------------------------- */

 idebug()
  /* Interactive debugging support */

  { boolean quit = FALSE;

    while (!quit)
```

```
    switch (upshift(resp(
         "I(dentifier, D(ebug level, A(ll symbols, C(ontinue? ")))
      {
        case 'I':  show_sym(); break;
        case 'A':  dump_all(); break;
        case 'D':  set_debug(); break;
        case '\0':
        case 'C':  quit = TRUE; break;
        default :  break;
       }
  } /* idebug */

/* ---------------------------------------------------------------------- */
```

```
 /* LR(1) PARSER apply procedure */

#include "CSPDEFS.C"
#include "CSPALL.C"
#include "CSPCODE.C"

extern boolean prlabchk(), check_mvar(), prochier();
extern int type_size(), dump_sem(), fixsize(), struct_assign(),
           incdum(), fixrsize();
extern symtabp declarit(), nohash(), enterrange(), enter_comp(),
               type_of_var();
extern semrecp gen_binnode(), bad_var(), gen_relnode(), gen_unary(),
               make_arnext(), make_consnode();

/* ---------------------------------------------------------------------- */

 apply(pflag, prodlen, tsemp)
  int pflag, prodlen;
  semrecp tsemp;

  { symtabp  t1symp, t2symp;
    semrecp t1semp, semptr;
    int ival;
    register int j,k;        /* temporary variables used */
    int msgcomm[MAXMESS];    /* to hold I/O command format */
    boolean defguard;        /* indicates presence of a default guard */

    /* Tagged productions are (in alphabetical order of flags):
     *
     *    1: SimpExp -> SimpExp + Term                     #ADDIT
     *    2: Arraydims -> Arraydims , CRange               #ADIM1
     *    3: AltStart ->                                   #ALTBEGIN
     *    4: AltComm -> [ GCommand ]                       #ALTER
     *    5: BoolTerm -> BoolTerm & BoolUnary              #ANDOP
     *    6: ArrayType -> [ Arraydims ] SimpleType         #ARRAYTYPE
     *    7: Extension -> [ BoolList ]                     #ARRID
     *    8: AssignComm -> TargetVar := Expression         #ASSIGN
     *    9: Guard -> BoolGuard                            #BGRD
     *    10: Guard -> BoolGuard ; InpGuard                #BIGRD
     *    11: BoolList -> BoolList , BoolExp               #BLIST1
     *    12: BoolList -> BoolExp                          #BLIST2
     *    13: Block -> ::                                  #BLOK
     *    14: SimpleType -> BOOLEAN                        #BTYP
     *    15: SimpleType -> BYTE                           #BYTTYP
     *    16: UnsigValu -> CHARCONS                        #CHRCNST
     *    17: SimpleType -> CHAR                           #CHTYP
     *    18: GCommand -> GCommand Nextg GuardComm         #COND1
     *    19: GCommand -> AltStart GuardComm               #COND2
     *    20: ConstItem -> IdentList = Constant ;          #CONSID
     *    21: Constructor -> <identifier> {                #CONSTRUCTOR
     *    22: CRange -> SignIntCons .. SignIntCons         #CRANGE
     *    23: SimpleType -> <identifier>                   #DECTYP
     *    24: GuardComm -> OTHERWISE Then CommandList #DEFGRD
     *    25: Relop -> =                                   #EQAL
     *    26: FieldList -> FieldList , Type                #FLIST1
     *    27: FieldList -> Type                            #FLIST2
     *    28: BoolGuard -> BoolGuard ; OneBool             #GLIST1
     *    29: BoolGuard -> OneBool                         #GLIST2
```

```
*    30: Relop -> >                                    #GRTR
*    31: Relop -> >=                                   #GTEQ
*    32: IdentList -> IdentList , <identifier>         #IDL1
*    33: IdentList -> <identifier>                     #IDL2
*    34: UnsigCons -> <identifier>                     #IDVAL
*    35: GuardComm -> Guard Then CommandList           #IFTHEN
*    36: Guard -> InpGuard                             #IGRD
*    37: UnsigInt -> <identifier>                      #INTID
*    38: SimpleType -> INTEGER                         #INTTYP
*    39: Labsub -> UnsigInt                            #LAB1
*    40: Labsub -> Range                               #LAB2
*    41: Extension -> ( BoolList )                     #LABID
*    42: Labelsubs -> Labelsubs , Labsub               #LABS
*    43: Relop -> <                                    #LESS
*    44: Relop -> <=                                   #LSEQ
*    45: StructExp -> Constructor BoolList }           #MESEXP
*    46: MessageItem -> <identifier> { FieldList } #MESITEM
*    47: StructTarget -> Constructor TargetVars } #MESSAGEVAR
*    48: Term -> Term % Unary                          #MODULO
*    49: Term -> Term * Unary                          #MULT
*    50: MultiGrd -> ( RangeList )                     #MULTIGD
*    51: Unary -> - Primary                            #NEGATE
*    52: Constant -> - UnsigCons                       #NEGCONS
*    53: SignIntCons -> - UnsigInt                     #NEGINT
*    54: Nextg -> []                                   #NEXTG
*    55: BoolUnary -> ~ BoolPri                        #NOTOP
*    56: Relop -> <>                                   #NTEQ
*    57: NullComm -> SKIP                              #NULLCOMM
*    58: BoolExp -> BoolExp | BoolTerm                 #OROP
*    59: ParCommand -> ParCommand // Process           #PARAL
*    60: Primary -> ( BoolExp )                        #PAREXP
*    61: ProcLabel -> _ <identifier>                   #PLAB1
*    62: ProcLabel -> _ <identifier> Labhead ( Labelsubs ) #PLAB2
*    63: ProcLabel ->                                  #PLAB3
*    64: ProcessName -> Simpvar                        #PNAME
*    65: Constant -> + UnsigCons                       #POSCONS
*    66: Labhead ->                                    #PRHEAD
*    67: Primary -> Simpvar                            #PRIVAR
*    68: Process -> ProcLabel Block CommandList        #PROCDECL
*    69: ParCommand -> Process                         #PROCESS
*    70: Term -> Term / Unary                          #QUOT
*    71: RangeList -> RangeList , Range               #RANGE1
*    72: RangeList -> Range                            #RANGE2
*    73: GuardComm -> MultiGrd Guard Then CommandList #RANGIFTHEN
*    74: Range -> <identifier> : CRange                #RDECL
*    75: InputComm -> Source ? TargetVar               #READIN
*    76: BoolPri -> SimpExp Relop SimpExp              #RELOP
*    77: Repeat -> *                                   #REPBEGIN
*    78: RepComm -> Repeat AltComm                     #REPT
*    79: SimpleType -> REAL                            #RTYP
*    80: SigTyp -> <identifier>                        #SGLTYP
*    81: StructExp -> Constructor }                    #SIGEXP
*    82: MessageItem -> <identifier> { }               #SIGITEM
*    83: StructTarget -> Constructor }                 #SIGNALVAR
*    84: StructCommand -> [ ParCommand ]               #STRUCTCOMM
*    85: Expression -> StructExp                       #STRUCTEXP
*    86: SimpExp -> SimpExp - Term                     #SUBT
*    87: Then -> --                                    #THEN
```

```
 *    88: TargetVars -> TargetVars , Simpvar      #TVAR1
 *    89: TargetVars -> Simpvar                   #TVAR2
 *    90: Varib -> <identifier>                   #VARH
 *    91: Simpvar -> Varib Extension              #VARID
 *    92: OneBool -> VarDeclarations BoolExp      #VGARD1
 *    93: OneBool -> BoolExp                      #VGARD2
 *    94: Guard -> VarDeclarations InpGuard       #VIGRD
 *    95: Varitem -> IdentList : Type ;           #VITEM
 *    96: OutputComm -> Destination ! Expression  #WRITEOUT
 *
 */

switch (pflag)
 {
  case ADDIT:   /* SimpExp -> SimpExp + Term */
    tsemp = gen_binnode( ADD_NODE);
    break;

  case ADIM1:   /* Arraydims -> Arraydims , CRange */
    tsemp = semstack[stackx-2];
    tlsymp = tsemp->usem.the_typep;
    while (tlsymp->usym.asymt.subtype != NULL)
        tlsymp = tlsymp->usym.asymt.subtype;
    tlsymp->usym.asymt.subtype = semstack[stackx]->usem.the_typep;
    break;

  case ALTER:   /* AltComm -> [ GCommand ] */
      /* remove last guard's local declarations */
    clearsym(plevel--,TRUE); semptr = semstack[stackx-1];
      /* reuse space allocated to guard vars */
    varadr = semptr->usem.bsemt.blexp->usem.altsemt.saveadr;
    gen_2byte(LOC,semptr->usem.bsemt.blexp->usem.altsemt.altlab4);
      /* location of start of jump table */
    gen_2byte(JBL,semptr->usem.bsemt.blexp->usem.altsemt.altlabl);
    defguard = FALSE;
    while (semptr)
      { if (semptr->usem.bsemt.blexp->usem.altsemt.altext == NULL)
          { if (semptr->usem.bsemt.blexp->usem.altsemt.usable)
              { gen_3byte( AJP,
                      semptr->usem.bsemt.blexp->usem.altsemt.altlab2,0);
                emitnl();
               } /* if a usable guard */
            else if (defguard) error("Only one otherwise guard allowed");
                  else
                    { defguard = TRUE;
                      gen_2byte( AJD,
                          semptr->usem.bsemt.blexp->usem.altsemt.altlab2);
                    }
          } /* if no extension */
         else jumprange(semptr->usem.bsemt.blexp->usem.altsemt.altext,
                        semptr->usem.bsemt.blexp->usem.altsemt.altlab2);
        semptr = semptr->usem.bsemt.blnext;
     }
    gen_2byte( LOC,
        semstack[stackx-1]->usem.bsemt.blexp->usem.altsemt.altlabl);
      /* location of end of jump table = altlabl */
    gen_1byte( EBL);
      /* if alt/rep command fails this code is executed */
```

```
  if (semstack[stackx-3]->semt == REP_NODE) /* break out of loop */
      gen_2byte( UJP, semstack[stackx-3]->usem.altsemt.altlab2);
    else  /* abort process if alt command fails initially
             and no otherwise guard present */
      if (!defguard) gen_1byte( EXT);
    /* now define label address for end of alt command = success */
  gen_2byte( LOC,
     semstack[stackx-1]->usem.bsemt.blexp->usem.altsemt.altlab3);
  if (semstack[stackx-3]->semt != REP_NODE)
    /* only for straight alt commands which succeed */
      gen_1byte(CRG); /* change guard ring */
  break;

case ALTBEGIN:  /* AltStart -> <empty> */
  plevel++;
    /* if enclosing guard always false, inner one too */
  if (gencode > 0) gencode++;
  tsemp =  newsem();
  tsemp->svtype = NULL;
  tsemp->semt = ALT_NODE;
  tsemp->usem.altsemt.altext = NULL;
  tsemp->usem.altsemt.usable = TRUE;
  tsemp->usem.altsemt.saveadr = varadr;  /* save present varadr */
    /* address of end of jump table = fail address */
  tsemp->usem.altsemt.altlab1 = incdum();
    /* first guard - this location */
  tsemp->usem.altsemt.altlab2 = incdum();
    /* address after jmp tbl -- that is, success */
  tsemp->usem.altsemt.altlab3 = incdum();
    /* address of start of jump table -- ALN transfer address */
  tsemp->usem.altsemt.altlab4 = incdum();
  gen_2byte( ALN, tsemp->usem.altsemt.altlab4);
  gen_2byte( LOC,tsemp->usem.altsemt.altlab2);
  gen_2byte( GDB, tsemp->usem.altsemt.altlab1);  /* start of guard */
  break;

case ANDOP:  /* BoolTerm -> BoolTerm & BoolUnary */
  tsemp = gen_binnode( AND_NODE);  break;

case ARRAYTYPE:  /* ArrayType -> ( SignIntCons .. SignIntCons ) Type */
  tsemp =  newsem();
  tsemp->semt = TYP_NODE;
  tsemp->svtype = NULL;  /* see later */
  t1symp = semstack[stackx-2]->usem.the_typep;
  t2symp = semstack[stackx]->usem.the_typep;
  if (t2symp->symt == USER_TYPE) t2symp = t2symp->usym.utypep;
  if ( t2symp->symt == MESS_TYPE || t2symp->symt == SIG_TYPE)
       { error( "Invalid base type for an array");
         t2symp = intptr;
       }
  tsemp->usem.the_typep = t1symp;
    /* recursively change sizes of array dimensions */
  t1symp->usym.asymt.size =  fixsize(t1symp,t2symp->usym.asymt.size);
  while (t1symp->usym.asymt.subtype != NULL)
      t1symp = t1symp->usym.asymt.subtype;
  t1symp->usym.asymt.subtype = t2symp;
  break;
```

```
case ARRID:  /* Extension -> [ BoolList ] */
  if (semstack[stackx-3]->usem.idsemt.symp->symt  == VARI)
    /* Varhead ok - construct a list of AR_EXT components based on
       the boollist */
     tsemp = make_arnext(semstack[stackx-1], AR_EXT,&ival);
   else error ("Invalid array name");
  break;

case ASSIGN:  /* AssignComm -> TargetVar := Expression */
  switch (struct_assign(semstack[stackx-2],semstack[stackx]))
   { case 0: gen_assign(semstack[stackx-2], semstack[stackx]);
             break;  /* unstructured assignment */
     case 1: /* message assignment */
             gen_sassign(semstack[stackx-2]->usem.gsemt.messexp,
                         semstack[stackx]->usem.gsemt.messexp);
             break;
     case 2: /* signal assignment */
             break;  /* no code emitted */
   }
 break;

case BGRD: /* Guard -> BoolGuard */
  tsemp = semstack[stackx];  break;

case BIGRD: /* Guard -> BoolGuard ; InpGuard */
  tsemp = semstack[stackx-2];  break;

case BLOK: /* Block -> :: */
  cur_proc->usym.psymt.locals = varadr - 1; /* gives stack required */
  gen_5byte(BEG, cur_proc->usym.psymt.procnum, cur_proc->rlevel,
        (cur_proc->usym.psymt.parproc) ?
           cur_proc->usym.psymt.parproc->usym.psymt.procnum : -1,
         cur_proc->usym.psymt.dim);
  tlsymp = cur_proc->usym.psymt.labptr;
  while (tlsymp != NULL)
    { gen_plab(tlsymp->usym.vsymt.vtypep);
      tlsymp = tlsymp->usym.vsymt.nxtvar;
    }
  tsemp =  newsem();
  tsemp->svtype = NULL;
  tsemp->semt = JP_NODE;
  tsemp->usem.jsemt.jplabl = incdum();
  tsemp->usem.jsemt.jplab2 = 0;
  gen_2byte(INC, tsemp->usem.jsemt.jplabl);
  break;

case TVAR1:  /* TargetVars -> TargetVars , Simpvar */
  if  (is_const(semstack[stackx]->semt) ||
        (semstack[stackx]->semt == STRNG) )
     semstack[stackx] = bad_var("Invalid use of constant",NULL);
      /* no break yet -- continue with next case */

case COND1:  /* GCommand -> GCommand Nextg GuardComm */
case RANGE1: /* RangeList -> RangeList, Range */
case BLIST1: /* BoolList -> BoolList , Boolean */
  tsemp = semstack[stackx-2];
  tlsemp = tsemp;
  while (tlsemp->usem.bsemt.blnext != NULL)
```

```
      tlsemp = tlsemp->usem.bsemt.blnext; /* find end of chain */
    semptr =  newsem();
    semptr->semt = BLIST_NODE;
    semptr->svtype = NULL;
    semptr->usem.bsemt.blnext = NULL;
    semptr->usem.bsemt.blexp = semstack[stackx];  /* catch semp node */
    tlsemp->usem.bsemt.blnext = semptr;
    break;

  case COND2:   /* GCommand -> AltStart GuardComm */
  case RANGE2: /* RangeList -> Range */
  case BLIST2: /* BoolList ->  Boolean */
    tsemp =  newsem();
    tsemp->semt = BLIST_NODE;
    tsemp->svtype = NULL;
    tsemp->usem.bsemt.blnext = NULL;
    tsemp->usem.bsemt.blexp = semstack[stackx]; /* catch semp node */
    break;

  case BTYP:   /* SimpleType -> BOOLEAN */
    tsemp =  newsem();
    tsemp->usem.the_typep = boolptr;
    tsemp->semt = TYP_NODE;
    break;

  case CHRCNST:   /* UnsigValu -> CHARCONS */
    tsemp = semstack[stackx];  break;

  case CONSID:   /* ConstItem -> IdentList = Constant ; */
    tlsemp = semstack[stackx-3]; /* catch idlist */
    while (tlsemp != NULL)
      { tlsymp = tlsemp->usem.insemt.idsymp;
        tlsymp = declarit(tlsymp,plevel,NULL); /* points at ST entry */
        tlsemp->usem.insemt.idsymp = tlsymp;
        tlsymp->symt = CONSVAR;
        semptr = semstack[stackx-1];
        tlsymp->usym.csymt.constyp = semptr->semt;
        switch (semptr->semt)
          { case FIXED:
            case BOOLS:
                tlsymp->usym.csymt.consival = semptr->usem.numval;
                break;
            case STRNG:
                tlsymp->usym.csymt.consival = semptr->usem.stx;
                break;
            case FLT:
                tlsymp->usym.csymt.consrval = semptr->usem.rval;
                break;
          }
        tlsemp = tlsemp->usem.insemt.idnext;
      }
    break;

  case CONSTRUCTOR:  /* Constructor -> <identifier> { */
    tsemp = semstack[stackx-1];
    tlsymp = tsemp->usem.idsemt.symp;
    if ( (tlsymp->symt != MESS_TYPE) && (tlsymp->symt != SIG_TYPE))
       {  if (tlsymp->symt != USER)
```

```
          { error("Invalid constructor");
            tlsymp = forcesym (tlsymp->sym, SIG_TYPE, GLOBLEV);
          }
         else { tlsymp->symt = SIG_TYPE; tlsymp->level = GLOBLEV; }
       tlsymp->usym.asymt.size = 0;
       tlsymp->usym.asymt.ldim = 0;
       tlsymp->usym.asymt.udim = msigadr++;
       tlsymp->usym.asymt.subtype = NULL;
     }
  break;

case CRANGE:   /* SignIntCons .. SignIntCons */
  /* forms a TYP_NODE semrec */
  tsemp =  newsem();
  tsemp->semt = TYP_NODE;
  tsemp->svtype = NULL;
  tsemp->usem.the_typep = enterrange(semstack[stackx]->usem.numval,
                            semstack[stackx-2]->usem.numval,ARRAY_TYPE);
  break;

case DECTYP:   /* SimpleType -> <identifier> */
  tsemp = newsem();
  copy_semrec(tsemp,semstack[stackx]);
  if ( (tsemp->usem.idsemt.symp->symt == USER_TYPE) ||
       (tsemp->usem.idsemt.symp->symt == MESS_TYPE) ||
       (tsemp->usem.idsemt.symp->symt == SIG_TYPE) )
      tlsymp = tsemp->usem.idsemt.symp;
    else { symerror(tsemp->usem.idsemt.symp->sym,
                   " invalid or unknown type");
           tlsymp = intptr; }
  tsemp->usem.the_typep = tlsymp;
  tsemp->semt = TYP_NODE;
  break;

case DEFGRD: /* GuardComm -> OTHERWISE -> CommandList */
  tsemp = semstack[stackx-3]; /* catch jp_node */
  tsemp->usem.altsemt.usable = FALSE;
  gen_2byte(EDF,tsemp->usem.altsemt.altlab3);
     /* jump to success or to interrupted communicating guard */
  if (gencode > 0) gencode--; /* restore code generation */
  break;

case EQAL:   /* Relop -> = */
  tsemp = gen_relnode(EQ_NODE); break;

case GLIST1:   /*  BoolGuard -> BoolGuard ; OneBool */
  tsemp = semstack[stackx-2];
  if (semstack[stackx] != NULL)
      gen_guard(semstack[stackx], tsemp->usem.altsemt.altlabl);
  break;

case GLIST2:   /*  BoolGuard -> OneBool */
  /* passes up a copy of the node with the relevant labels */
  tsemp = semstack[stackx-1];
  if (semstack[stackx] != NULL)
      gen_guard(semstack[stackx], tsemp->usem.altsemt.altlabl);
  break;
```

```
case GTEQ:  /* Relop -> >= */
  tsemp = gen_relnode(GEQ_NODE);  break;

case GRTR:  /* Relop -> > */
  tsemp = gen_relnode(GTR_NODE);  break;

case LABS:   /* Labelsubs -> Labelsubs , Labsub */
case FLIST1: /* FieldList -> FieldList , Type */
case IDL1:   /* IdentList -> IdentList , <identifier> */
  tsemp = semstack[stackx-2]; /* catch Identlist */
  tlsemp = tsemp;
  while (tlsemp->usem.insemt.idnext != NULL)
         tlsemp = tlsemp->usem.insemt.idnext; /* go to end of chain */
  semptr =  newsem();
  semptr->semt = IDLIST_NODE;
  semptr->svtype = NULL;
  semptr->usem.insemt.idnext = NULL;
  switch (pflag)
   { case IDL1:
       semptr->usem.insemt.idsymp = semstack[stackx]->usem.idsemt.symp;
       break;
     case LABS:
       semptr->usem.insemt.idsymp = semstack[stackx]->usem.insemt.idsymp;
       break;
     case FLIST1:
       semptr->usem.insemt.idsymp = semstack[stackx]->usem.the_typep;
       break;
   }
  tlsemp->usem.insemt.idnext = semptr;
  break;

case FLIST2: /* FieldList -> Type */
case IDL2:   /* IdentList -> <identifier> */
  tsemp =  newsem();
  tsemp->svtype = NULL;
  tsemp->semt = IDLIST_NODE;
  tsemp->usem.insemt.idsymp = (pflag == IDL2) ?
          semstack[stackx]->usem.idsemt.symp  :
          semstack[stackx]->usem.the_typep;
  tsemp->usem.insemt.idnext = NULL; /* end of chain */
  break;

case IDVAL:  /* UnsigCons -> <identifier> */
  tlsymp = semstack[stackx]->usem.idsemt.symp;
  if (tlsymp->symt != CONSVAR)
       { error("Constant value expected");
         tsemp =  newsem();
         tsemp->semt = FIXED; tsemp->svtype = intptr;
         tsemp->usem.numval  = 1; }
    else tsemp = make_consnode(tlsymp);
  break;

 case RANGIFTHEN: /* GuardComm -> MultiRange Guard Then CommandList */
 case IFTHEN:     /* GuardComm -> Guard Then CommandList */
  tsemp = semstack[stackx-2];  /* return a jp_node */
  gen_2byte(UJP, tsemp->usem.altsemt.altlab3); /* jump to success */
  if (gencode > 0) gencode--; /* restore code generation */
  break;
```

```
 case IGRD:  /* Guard -> InpGuard */
  tsemp = semstack[stackx-1];  break;

 case INTID:  /* UnsigInt -> <identifier> */
  tsemp =  newsem();
  if (semstack[stackx]->usem.idsemt.symp->symt == CONSVAR &&
      (semstack[stackx]->usem.idsemt.symp->usym.csymt.constyp == FIXED
       || semstack[stackx]->usem.idsemt.symp->usym.csymt.constyp
             == BOOLS) )
      ival = semstack[stackx]->usem.idsemt.symp->usym.csymt.consival;
    else { error(" Integer constant expected");
           ival = 1; }
  tsemp->semt = semstack[stackx]->usem.idsemt.symp->usym.csymt.constyp;
  tsemp->svtype = intptr;
  tsemp->usem.numval = ival;
  break;

case BYTTYP:  /* SimpleType -> BYTE */
case CHTYP:   /* SimpleType -> CHAR */
  tsemp =  newsem();
  tsemp->usem.the_typep = charptr;
  tsemp->semt = TYP_NODE;
  break;

case INTTYP:  /* SimpleType -> INTEGER */
  tsemp =  newsem();
  tsemp->usem.the_typep = intptr;
  tsemp->semt = TYP_NODE;
  break;

case LAB1:  /* Labsub -> UnsigInt */
  /* NB. should not increase varadr for one of these dummy vars, as
     these can never be referenced; but it is done for ease of
     interpretation. */
  t1symp = enterrange(semstack[stackx]->usem.numval,
           semstack[stackx]->usem.numval,SUBRANGE);
  t1symp->usym.asymt.subtype = intptr;
  t2symp = nohash(VARI, "<proclabel>");
  t2symp->usym.vsymt.saddr = varadr;
  varadr += INT_SIZE;
  t2symp->usym.vsymt.canchange = FALSE;
  t2symp->usym.vsymt.vtypep = t1symp;
  tsemp = newsem();
  tsemp->semt = IDLIST_NODE;
  tsemp->svtype = NULL;
  tsemp->usem.insemt.idnext = NULL;
  tsemp->usem.insemt.idsymp = t2symp;
  break;

case LAB2:  /* Labsub -> Range */
  semptr = semstack[stackx];  /* catch IDENT node */
  tsemp =  newsem();
  tsemp->semt = IDLIST_NODE;
  tsemp->svtype = NULL;
  tsemp->usem.insemt.idnext = NULL;
  tsemp->usem.insemt.idsymp = semptr->usem.idsemt.symp;
  break;
```

```
case LABID:  /* Extension -> ( BoolList ) */
  ival = 0;
  tlsymp = semstack[stackx-3]->usem.idsemt.symp;
  if (tlsymp->symt == PROC_TYPE)
      /* construct a list of PROC_EXT components */
       tsemp = make_arnext(semstack[stackx-1], PROC_EXT,&ival);
    else error ("Invalid process label");
  if (tlsymp->usym.psymt.dim >= 0 && tlsymp->usym.psymt.dim != ival)
       error("Process label required or invalid");
  if ( !tlsymp->usym.psymt.defined) tlsymp->usym.psymt.dim = ival;
  break;

case LSEQ:  /* Relop -> <= */
  tsemp = gen_relnode(LEQ_NODE);  break;

case LESS:  /* Relop -> < */
  tsemp = gen_relnode(LES_NODE);  break;

case MESITEM:  /* MessageItem -> <identifier> { FieldList } */
  tlsemp = semstack[stackx-3];
  tlsymp = declarit(tlsemp->usem.idsemt.symp, GLOBLEV, NULL);
  tlsymp->symt = MESS_TYPE;
  tlsymp->usym.asymt.ldim = 0;  /* number of components */
  tlsymp->usym.asymt.size = 0;
  tlsymp->usym.asymt.udim = msigadr++;  /* unique number */
  tlsymp->usym.asymt.subtype =
      enter_comp(semstack[stackx-1], &tlsymp->usym.asymt.ldim,
                 &tlsymp->usym.asymt.size,  0);
  if (tlsymp->usym.asymt.ldim > MAXMESS / 2 )
    { error ("Too many message components");
      tlsymp->usym.asymt.ldim = MAXMESS - 1;
    }
  break;

case MESEXP:     /* StructExp -> Constructor BoolList } */
case MESSAGEVAR: /* StructTarget -> Constructor  TargetVars } */
  if (semstack[stackx-2] != NULL)
    { tlsymp = semstack[stackx-2]->usem.idsemt.symp;
      if (tlsymp->symt != MESS_TYPE)
        error("Identifier not a message constructor");
    }
   else tlsymp = NULL;
  tsemp =  newsem();
  tsemp->semt = MESS_NODE;
  tsemp->svtype = NULL;
  tsemp->usem.gsemt.messexp = semstack[stackx-1]; /* catch blist */
  tsemp->usem.gsemt.messvar = tlsymp;
  if ( (pflag == MESSAGEVAR) &&  (tlsymp != NULL) &&
     ( ! check_mvar(tlsymp->usym.asymt.subtype, semstack[stackx-1])) )
      error("Message use not compatible with declaration");
  break;

case MODULO:  /* Term -> Term % Unary */
  tsemp = gen_binnode(MOD_NODE);  break;

case MULT:  /* Term -> Term * Unary */
  tsemp = gen_binnode(MPY_NODE);  break;
```

```
case MULTIGD: /* MultiGrd -> ( RangeList ) */
    /* Rangelist is passed as a BLIST node */
  tsemp = semstack[stackx-3];  /* get jp_node */
  semptr = semstack[stackx-1];
  tsemp->usem.altsemt.altext = semptr;
  while (semptr != NULL)
    { gen_var(semptr->usem.bsemt.blexp, STI);
      semptr = semptr->usem.bsemt.blnext;
    }
  break;

case NEGATE:  /* Unary -> - Primary */
  tsemp = gen_unary(NEG_NODE);  break;

case NEGCONS:  /* Constant -> - UnsigCons */
  tsemp = semstack[stackx];
  if (tsemp->semt == FIXED) tsemp->usem.numval = - tsemp->usem.numval;
    else if (tsemp->semt == FLT) tsemp->usem.rval = - tsemp->usem.rval;
     else error("Cannot negate non-numerical values");
  break;

case NEGINT:  /* SignIntCons -> - UnsigInt */
  tsemp = semstack[stackx];
  switch (tsemp->semt)
    { case BOOLS: error("Cannot negate non-numerical values");
                  break;
      case FIXED:
       tsemp->usem.numval = - tsemp->usem.numval; break;
    }
  break;

case NTEQ:  /* Relop -> <> */
  tsemp = gen_relnode(NEQ_NODE);  break;

case NEXTG:  /* Nextg -> [ ] */
  tsemp =  newsem();
  tsemp->svtype = NULL;
  tsemp->semt = ALT_NODE;
  tsemp->usem.altsemt.altext = NULL;
  tsemp->usem.altsemt.usable = TRUE;
  semptr = semstack[stackx-1]->usem.bsemt.blexp;  /* Altbeg record */
    /* get jump table address = fail address */
  tsemp->usem.altsemt.altlabl = semptr->usem.altsemt.altlabl;
  tsemp->usem.altsemt.altlab2 = incdum(); /* next guard address */
    /* get success address of alternate stmt */
  tsemp->usem.altsemt.altlab3 = semptr->usem.altsemt.altlab3;
  tsemp->usem.altsemt.altlab4 = semptr->usem.altsemt.altlab4;
  tsemp->usem.altsemt.saveadr = semptr->usem.altsemt.saveadr;
  gen_2byte(LOC, tsemp->usem.altsemt.altlab2);
  gen_2byte(GDB, tsemp->usem.altsemt.altlabl);
    /* now remove previous guard's local declarations and
       reuse stack space */
  clearsym(plevel,TRUE); varadr = semptr->usem.altsemt.saveadr;
  break;

case NOTOP:  /* BoolUnary -> ~ BoolPri */
  tsemp = gen_unary(NOT_NODE);  break;
```

```
case NULLCOMM:  /* NullComm -> SKIP */
  gen_1byte(NOP);  break;

case OROP:  /* BoolExp -> BoolExp | BoolTerm */
  tsemp = gen_binnode(OR_NODE);  break;

case PARAL:  /* ParCommand -> ParCommand // Process */
  tsemp = semstack[stackx-2];  /* save node with parent process info */
  cur_proc = tsemp->usem.idsemt.symp;
  break;

case PAREXP:  /* Primary -> ( BoolExp ) */
  tsemp = semstack[stackx-1];  /* just catch the boolean */
  break;

case PLAB1:  /* ProcLabel -> <identifier>  */
  plevel++; rlevel++; varadr = FIRSTADR;
  tlsymp = cur_proc;    /* save parent process */
    /* process sym entry at global level 0 */
  cur_proc = declarit(semstack[stackx]->usem.idsemt.symp,0,tlsymp);
  enterproc(cur_proc);
  if (cur_proc->usym.psymt.dim < 0) cur_proc->usym.psymt.dim = 0;
    else if (cur_proc->usym.psymt.dim)  /* dimension not zero */
          { error("Process definition inconsistent ");
            cur_proc->usym.psymt.dim = 0;
          }
  cur_proc->usym.psymt.defined = TRUE;
  cur_proc->usym.psymt.parproc = tlsymp;
  break;

case PLAB2:  /* ProcLabel -> <identifier> Labhead ( Labelsubs ) */
  enterproc(cur_proc);
  cur_proc->usym.psymt.defined = TRUE;
    /* parent process returned by labhead */
  cur_proc->usym.psymt.parproc = semstack[stackx-3]->usem.idsemt.symp;
  tlsemp = semstack[stackx-1];
  cur_proc->usym.psymt.labptr = tlsemp->usem.insemt.idsymp;
  k = 0;  j = 1;
  while (tlsemp != NULL)
    { if (++k > MAXDIM) /* another dimension */
            { error("Too many process labels");  k--;  }
      tlsymp = tlsemp->usem.insemt.idsymp;
      if (tlsemp->usem.insemt.idnext == NULL)
           tlsymp->usym.vsymt.nxtvar = NULL;
       else tlsymp->usym.vsymt.nxtvar =
              tlsemp->usem.insemt.idnext->usem.insemt.idsymp;
      t2symp = tlsymp->usym.vsymt.vtypep;
      j = j * (t2symp->usym.asymt.size);
      tlsemp = tlsemp->usem.insemt.idnext;
     }
    /* now recursively change the sizes of the subrange nodes */
  tlsymp = cur_proc->usym.psymt.labptr;
  tlsymp->usym.vsymt.vtypep->usym.asymt.size = fixrsize(tlsymp, 1);
  if ( (cur_proc->usym.psymt.dim >= 0) &&
       (cur_proc->usym.psymt.dim != k))
            error("Process definition inconsistent");
    else if (!prlabchk(semstack[stackx-4]->usem.idsemt.symp, cur_proc))
```

```
        error("Process labels overlap with previous process definition");
    cur_proc->usym.psymt.dim = k;
    cur_proc->usym.psymt.noprocs = j;  /* no. of instances */
    break;

  case PLAB3:    /* ProcLabel -> <empty> */
    plevel++; rlevel++; varadr = FIRSTADR;
    tlsymp = cur_proc;
    cur_proc = forcesym("<process>", USER, 0);
    enterproc(cur_proc);
    cur_proc->usym.psymt.dim = 0;
    cur_proc->usym.psymt.defined = TRUE;
    cur_proc->usym.psymt.parproc = tlsymp;
    break;

  case PNAME:   /* ProcessName -> SimpVar */
    tsemp = semstack[stackx];
    if ((tsemp->semt != IDENT) ||
        (tsemp->usem.idsemt.symp->symt != PROC_TYPE))
      /* report and patch an error */
       { error(" Invalid process name");
         tlsymp = forcesym(errsym, PROC_TYPE, 0);
         tsemp->usem.idsemt.symp = tlsymp;
         tsemp->svtype = NULL;
         enterproc(tlsymp);
        }
      else if ( prochier(tsemp->usem.idsemt.symp, cur_proc) )
                error("I/O invalid with current process hierarchy");
    break;

  case POSCONS:  /* Constant -> + UnsigCons */
    tsemp = semstack[stackx];
    if (( tsemp->semt != FIXED) && (tsemp->semt != FLT) )
       error("Syntax error");
    break;

  case PRHEAD:  /* Labhead  -> <empty> */
    /* Passes back an IDENT node containing a pointer to the
       parent process */
    plevel++; rlevel++;
    varadr = FIRSTADR;
    tsemp =  newsem();
    tsemp->semt = IDENT;
    tsemp->usem.idsemt.symp = cur_proc;
    tsemp->usem.idsemt.vext = NULL;
    cur_proc = declarit(semstack[stackx]->usem.idsemt.symp, 0,
               tsemp->usem.idsemt.symp);
    break;

  case PRIVAR:  /* Primary -> Simpvar */
    tsemp = semstack[stackx];
    if (tsemp->semt == IDENT)
      { /* Simpvar is an identifier */
        tlsymp = semstack[stackx]->usem.idsemt.symp;
        if (tlsymp->symt != VARI)  /* report and patch an error */
          { symerror(tlsymp->sym, " Unknown variable type ");
            tlsymp = forcesym(errsym, VARI, plevel);
            tsemp->usem.idsemt.symp = tlsymp;
```

```
          tsemp->svtype = intptr;
          tlsymp->usym.vsymt.saddr = 1;
          tlsymp->usym.vsymt.vtypep = intptr;
          tlsymp->usym.vsymt.canchange = TRUE;
          tlsymp->usym.vsymt.nxtvar = NULL;
        }
    }
  break;

case PROCDECL:  /* Process -> ProcLabel Block CommandList */
  clearsym(plevel--, FALSE); rlevel--;
  gen_3byte(DEF, semstack[stackx-1]->usem.jsemt.jplabl,
            cur_proc->usym.psymt.locals+1);  /* entire inc value */
  gen_1byte(EXT /*, cur_proc->usym.psymt.locals+1*/);
  break;

case PROCESS:  /* ParCommand -> Process */
  tsemp =  newsem();
  tsemp->svtype = NULL;
  tsemp->semt = IDENT;
    /* keep track of parent process */
  tsemp->usem.idsemt.symp = cur_proc->usym.psymt.parproc;
  tsemp->usem.idsemt.vext = NULL;
  cur_proc = cur_proc->usym.psymt.parproc;
  break;

case QUOT:  /* Term -> Term / Unary */
  tsemp = gen_binnode(QUOT_NODE);  break;

case RDECL:  /* Range -> <identifier> : CRange */
   /* passes an IDENT node up */
  tsemp = semstack[stackx-2];
  tsemp->usem.idsemt.symp =
           declarit(tsemp->usem.idsemt.symp, plevel,NULL);
  tsemp->svtype = intptr;
  tlsymp = semstack[stackx]->usem.the_typep;
  tlsymp->usym.asymt.subtype = intptr;
  tlsymp->symt = SUBRANGE; strcpy(tlsymp->sym,"<subrange>");
  entervar(tsemp->usem.idsemt.symp, tlsymp, FALSE);
  if (varadr-1 > cur_proc->usym.psymt.locals)
   cur_proc->usym.psymt.locals = varadr - 1; /* gives stack required */
  break;

case WRITEOUT:  /* OutputComm -> Destination ! Expression */
case READIN:    /* InputComm -> Source ? TargetVar */
  ival = 0;
  if (semstack[stackx]->semt == MESS_NODE)
    { msgcomm[ival++] = MSS;
      msgcomm[ival++] = semstack[stackx]->usem.gsemt.messvar ?
           semstack[stackx]->usem.gsemt.messvar->usym.asymt.udim :
           NOSTRUCT;
      readwrite( (pflag == READIN),
                 semstack[stackx]->usem.gsemt.messexp, msgcomm, &ival);
    }
   else gen_rw( (pflag == READIN), semstack[stackx], 0, msgcomm, &ival);
  gen_procnum(semstack[stackx-2]);
  gen_1byte((pflag == READIN) ? MSI : MSO);
  for (j=0;j<ival; j++) genbyte(msgcomm[j]);
```

```
  genbyte(-1);  /* used to terminate msg command string */
  emitnl();
  break;

case RELOP:  /* BoolPri -> SimpExp Relop SimpExp */
  tsemp = gen_binnode(semstack[stackx-1]->semt);  break;

case REPT:  /* RepComm -> Repeat AltComm */
  gen_2byte(UJP, semstack[stackx-1]->usem.jsemt.jplab1);
  gen_2byte(LOC, semstack[stackx-1]->usem.jsemt.jplab2);
  break;

case REPBEGIN:  /* RepComm -> *  */
  tsemp =  newsem();
  tsemp->svtype = NULL;
  tsemp->semt = REP_NODE;
  tsemp->usem.jsemt.jplab1 = incdum(); /* start of repeat stmt */
    /* next instr after repeat stmt */
  tsemp->usem.jsemt.jplab2 = incdum();
  gen_2byte(LOC, tsemp->usem.jsemt.jplab1);
  break;

case RTYP:  /* SimpleType -> REAL */
  tsemp =  newsem();
  tsemp->usem.the_typep = realptr;
  tsemp->semt = TYP_NODE;
  break;

case SGLTYP:  /*  SigItem -> <identifier> */
  /* declares global variables of signals with constructors */
case SIGITEM:  /* MessageItem -> <identifier> { } */
  /* declares global signal types */
  t1semp = (pflag == SIGITEM) ? semstack[stackx-2] : semstack[stackx];
  t1symp = declarit(t1semp->usem.idsemt.symp, GLOBLEV, NULL);
  t1symp->symt = SIG_TYPE;
  t1symp->usym.asymt.ldim = 0;
  t1symp->usym.asymt.udim = msigadr++; /* unique constructor num */
  t1symp->usym.asymt.subtype = NULL;
  t1symp->usym.asymt.size = 0;
  if (pflag == SIGITEM) break;

    /* now create variable entry in symbol table */
  t2symp = forcesym(t1symp->sym,VARI,GLOBLEV);
  t2symp->usym.vsymt.vtypep = t1symp;
  t2symp->usym.vsymt.saddr = 0;
  t2symp->usym.vsymt.nxtvar = NULL;
  t2symp->usym.vsymt.canchange = TRUE;
  break;

case SIGEXP:    /* StructExp -> Constructor } */
case SIGNALVAR: /* StructTarget -> Constructor  } */
  if (semstack[stackx-1] != NULL)
   { t1symp = semstack[stackx-1]->usem.idsemt.symp;
     if (t1symp->symt != SIG_TYPE)
          error("Identifier is not a signal variable");
   }
   else t1symp = NULL;
  tsemp =  newsem();
```

```
  tsemp->semt = SIG_NODE;
  tsemp->svtype = NULL;
  tsemp->usem.sigsemt.sigvar = tlsymp;
  break;

case STRUCTCOMM:   /* StructCommand -> [ ParCommand ] */
  cur_proc = semstack[stackx-1]->usem.idsemt.symp;
  varadr = cur_proc->usym.psymt.locals+1;
  break;

case SUBT:   /* SimpExp -> SimpExp - Term */
  tsemp = gen_binnode(SUB_NODE);  break;

case THEN:   /* Then -> -- */
  gen_lbyte(GDE);  break;

case TVAR2:   /* TargetVars  -> TargetVar */
  tsemp =  newsem();
  tsemp->semt = BLIST_NODE;
  tsemp->svtype = NULL;
  if (is_const(semstack[stackx]->semt) ||
      semstack[stackx]->semt == STRNG )
   { semstack[stackx] = bad_var("Cannot assign to constants",NULL);
     semstack[stackx]->svtype = intptr;
   }
  tsemp->usem.bsemt.blexp = semstack[stackx];
  tsemp->usem.bsemt.blnext = NULL;
  break;

case VGARD1:   /* VarGuard -> VarDeclarations BoolExp */
case VGARD2:   /* VarGuard -> BoolExp */
  tsemp = semstack[stackx]; /* pass expression up */
  break;

case VARH:   /* Varib -> <identifier> */
  tsemp = semstack[stackx];
  if (tsemp->usem.idsemt.symp->symt == CONSVAR)
      tsemp = make_consnode(tsemp->usem.idsemt.symp);
   else if (tsemp->usem.idsemt.symp->symt == USER)
         tsemp = bad_var (" Undeclared identifier - assumed process",
                         tsemp->usem.idsemt.symp);
    else if ( (tsemp->usem.idsemt.symp->symt != VARI) &&
              (tsemp->usem.idsemt.symp->symt != PROC_TYPE) )
           tsemp = bad_var (" Invalid variable type ",NULL);
  break;

case VARID:   /* Simpvar -> Varib Extension */
  tsemp = semstack[stackx-1];
  if (tsemp->semt == IDENT)    /* Varhead - an IDENT type */
   { tlsymp = tsemp->usem.idsemt.symp;
       /* Vext may or may not be NULL */
     tsemp->usem.idsemt.vext = semstack[stackx];
     if (semstack[stackx] == NULL && tlsymp->symt == PROC_TYPE)
        { if (tlsymp->usym.psymt.dim > 0)
             error("Process label required or invalid");
          tlsymp->usym.psymt.dim = 0;
        }
    tsemp->svtype = type_of_var(tsemp);
```

```
        if (tsemp->svtype == NULL)
            /* process label or type_of_var found an error */
          { if (tsemp->usem.idsemt.symp->symt != PROC_TYPE)
                tsemp->svtype = intptr;
            tsemp->usem.idsemt.vext = NULL;
          }
        if (tsemp->svtype->symt == MESS_TYPE)
          { semptr =  newsem();
            semptr->svtype = NULL;
            semptr->semt = MESS_NODE;
            semptr->usem.gsemt.messvar = tsemp->svtype;
            semptr->usem.gsemt.messexp = tsemp;
            tsemp = semptr;
          }
         else if (tsemp->svtype->symt == SIG_TYPE)
          { semptr =  newsem();
            semptr->svtype = NULL;
            semptr->semt = SIG_NODE;
            semptr->usem.sigsemt.sigvar = tsemp->svtype;
            tsemp = semptr;
          }
       }
      break;

    case VIGRD:  /* Guard -> VarDeclarations InpGuard */
      tsemp = semstack[stackx-2];  break;

    case VITEM:  /* Varitem -> IdentList : Type ; */
      tlsemp = semstack[stackx-3];
      while (tlsemp != NULL)
       {  /* enter new local var into symbol table - complain if
               already declared */
          tlsemp->usem.insemt.idsymp  =
                   declarit(tlsemp->usem.insemt.idsymp, plevel,NULL);
          entervar(tlsemp->usem.insemt.idsymp,
                   semstack[stackx-1]->usem.the_typep, TRUE);
          tlsemp = tlsemp->usem.insemt.idnext;
       }
      if (varadr-1 > cur_proc->usym.psymt.locals)
         /* gives stack required for local variables and return adr */
             cur_proc->usym.psymt.locals = varadr - 1;
      break;

    default: break;

   }  /* apply switch */
  appsemp = tsemp; /* enable new tsemp value to be seen by parser() */
 }  /* apply */

/* ---------------------------------------------------------------------- */
```

```
  /* LR1 semantic procedures */

#include "CSPDEFS.C"
#include "CSPALL.C"
#include "CSPCODE.C"

extern int check_compat();

/* ------------------------------------------------------------------- */

 int incdum()
  /* Increments the dumlabx label; and checks for too many labels */

  { if (dumlabx == LABELMAX)
         warning("Program contains too many indirect jumps");
    return(dumlabx++);
  }  /* incdum */

/* ------------------------------------------------------------------- */

 boolean math(tag)
  semtype tag;

  { return ( (tag >= FIXED) && (tag <= NEG_NODE) ); }

/* ------------------------------------------------------------------- */

 boolean boo(tag)
  semtype tag;

  { return ( (tag >= BOOLS) && (tag <= NEQ_NODE) ); }

/* ------------------------------------------------------------------- */

 boolean arithtypes(tag)
  semtype tag;
   /* Test whether semantic node is one of arithmetical or boolean */

   { return ( (tag >= FIXED) && (tag <= NEQ_NODE) ); }

/* ------------------------------------------------------------------- */

 symtype utype(t1, t2)
  symtype t1, t2;
   /* Returns bool_type if either operand is boolean, int_type if both are
      integer or character, otherwise returns real_type */

   {  if (t1 == BOOL_TYPE || t2 == BOOL_TYPE) return(BOOL_TYPE);
      return ( ((t1 == INT_TYPE || t1 == CHAR_TYPE) &&
               (t2 == INT_TYPE || t2 == CHAR_TYPE)) ? INT_TYPE : REAL_TYPE);
   }  /* utype */

/* ------------------------------------------------------------------- */

 boolean is_std(st)
  symtype st;
   /* Test whether a given symtype is a standard type */
```

```
  { return ( (st == INT_TYPE || st == REAL_TYPE || st == BOOL_TYPE ||
             st == CHAR_TYPE) );  }

/* ---------------------------------------------------------------------- */

 boolean is_const(st)
  semtype st;
   /* Test whether a given semtype is constant */

   {  return ( ((st == FIXED) || (st == FLT) || (st == BOOLS)) ); }

/* ---------------------------------------------------------------------- */

 copy_semrec(dst, src)
  semrecp src, dst;
   /* Copies contents of one semantic record to another */

  { char *s = (char *) src, *t = (char *) dst;
    int size = sizeof(struct semrec);
    register int i = 0;

    while (i++ <= size) *t++ = *s++;
  }  /* copy_semrec */

/* ---------------------------------------------------------------------- */

 boolean check_type(tag,vtype)
  symtype vtype;
  semtype tag;
   /* Looks at an operand (sptr) and makes sure that it is compatible with
      the operator (tag) */

  { switch (vtype)
      { case BOOL_TYPE:  if ( (tag >= AND_NODE) && (tag <= NEQ_NODE) )
                               return(TRUE);
                         break;
        case REAL_TYPE:
        case CHAR_TYPE:
        case INT_TYPE:   if ( ((tag >= EQ_NODE) && (tag <= NEQ_NODE))
                                || math(tag) ) return (TRUE);
                         break;
        default: break;
      }
    error("Invalid type for operation");
    return(FALSE);
  }  /* check_type */

/* ---------------------------------------------------------------------- */

 symtabp nohash(sstype, ssym)
  symtype sstype;
  char ssym[];
   /* Creates a symbol table entry, which is not linked into the hash chains.
      Used for entries describing type extensions and message variables. */

    { symtabp tl;

        tl = newsym();  /* new ST entry */
```

```
      tl->next = NULL;
      tl->symt = sstype;
      strcpy(tl->sym, ssym);  /* so it has a default sym */
      tl->level = plevel;
      tl->rlevel = rlevel;
      return(tl);
    }  /* nohash */

/* ------------------------------------------------------------------------ */

entervar(symp,typtr,assign)
 symtabp symp,typtr;
 boolean assign;
  /* Fills in the default values for a variable's symbol table entry */

 { symp->symt = VARI;
   symp->usym.vsymt.saddr = varadr;
   if (typtr->symt == USER_TYPE) typtr = typtr->usym.utypep;
     /* moves indirect type reference to direct record ref */
   symp->usym.vsymt.vtypep = typtr;
   symp->usym.vsymt.canchange = assign;
   symp->usym.vsymt.nxtvar = NULL;
   varadr = varadr + type_size(typtr); /* next location */
 }  /* entervar */

/* ------------------------------------------------------------------------ */

symtabp declarit(sp,vlevel,parent)
 symtabp sp,parent;
 int vlevel;
  /* This forces a new symbol entry if the existing one is not of type
     USER - the default type inserted by the lexical analyser */

 { symtabp tp;

   if (sp->symt == USER) { sp->level = vlevel;
                           sp->rlevel = rlevel;
                           return(sp); }
     else
      if (sp->symt == PROC_TYPE)
        { /* takes care of processes */
          if (!sp->usym.psymt.defined) return(sp);
            else
             if (sp->rlevel != rlevel || sp->usym.psymt.parproc != parent)
              { symerror(sp->sym,
                       "  defined previously with different parent");
                return(forcesym(sp->sym, USER, 0)); }
              else
                { tp = nohash(PROC_TYPE, sp->sym);
                  tp->level = 0;
                  tp->usym.psymt.procnum = sp->usym.psymt.procnum;
                  tp->usym.psymt.parproc = sp->usym.psymt.parproc;
                  tp->usym.psymt.dim = sp->usym.psymt.dim;
                  while (sp->usym.psymt.nxtproc)
                    sp = sp->usym.psymt.nxtproc;
                      /* find end of process entry chain */
                  return(sp->usym.psymt.nxtproc = tp);
                }
```

```
        } /* process entries */
      else /* make a new entry */
        { if (sp->level == vlevel)
            symerror(sp->sym, " previously declared");
          return(forcesym(sp->sym, USER, vlevel));
        }
  }  /* declarit */

/* ------------------------------------------------------------------------ */

 int type_size(sp)
  symtabp sp;
   /* Given a symbol table reference, this function returns the size of
      the reference */

  { switch (sp->symt)
     { case VARI: return(sp->usym.vsymt.vtypep->usym.asymt.size);
                  break;
       case USER_TYPE: return(type_size(sp->usym.utypep));
                       break;
       case INT_TYPE:
       case CHAR_TYPE:
       case BOOL_TYPE:
       case REAL_TYPE:
       case MESS_TYPE:
       case SIG_TYPE:
       case ARRAY_TYPE: return(sp->usym.asymt.size); break;
       case SUBRANGE: return(type_size(sp->usym.asymt.subtype)); break;
       default: symerror(sp->sym, " invalid type");
                return(1);
     }
  }  /* type_size */

/* ------------------------------------------------------------------------ */

 static boolean duplabs(oldpr, newpr)
  symtabp oldpr, newpr;
   /* Checks for duplicate process labels; no checks made for dimension
      consistency as this is done later */

 {  int usiz1, lsiz1, usiz2, lsiz2;

    if (!oldpr || !newpr) return(TRUE); /* all dimensions overlap */
    usiz1 = oldpr->usym.vsymt.vtypep->usym.asymt.udim;
    lsiz1 = oldpr->usym.vsymt.vtypep->usym.asymt.ldim;
    usiz2 = newpr->usym.vsymt.vtypep->usym.asymt.udim;
    lsiz2 = newpr->usym.vsymt.vtypep->usym.asymt.ldim;
    if ( (usiz1 <= usiz2 && usiz1 >= lsiz2) ||
         (lsiz1 >= lsiz2 && lsiz1 <= usiz2) ||
         (usiz2 <= usiz1 && usiz2 >= lsiz1) ||
         (lsiz2 >= lsiz1 && lsiz2 <= usiz1))
        /* found a duplicate; now check other dimensions */
       return(duplabs(oldpr->usym.vsymt.nxtvar, newpr->usym.vsymt.nxtvar));
     else return(FALSE);
  }  /* duplabs */
```

```
/* ------------------------------------------------------------------ */

 boolean prlabchk(sp, newpr)
 symtabp sp, newpr;
  /* Checks that instance labels of multi-defined processes do not overlap */

 {  if (sp == newpr) return(TRUE);
      else if (duplabs(sp->usym.psymt.labptr, newpr->usym.psymt.labptr))
               return(FALSE);
            else return(prlabchk(sp->usym.psymt.nxtproc, newpr));
 }  /* prlabchk */

/* ------------------------------------------------------------------ */

 int fixsize(ptr,sz)
  symtabp ptr;
  int sz;
   /* Recursively change the sizes of array type nodes */

   { if (ptr == NULL) return(sz);
       else
         return( (ptr->usym.asymt.size = fixsize(ptr->usym.asymt.subtype,sz)
                  * ptr->usym.asymt.size) );
   } /* fixsize */

/* ------------------------------------------------------------------ */

 int fixrsize(ptr,sz)
  symtabp ptr;
  int sz;
   /* Recursively change the sizes of subrange type nodes */

  { if (ptr == NULL) return(sz);
     else
        return( (ptr->usym.vsymt.vtypep->usym.asymt.size =
                 fixrsize(ptr->usym.vsymt.nxtvar,sz)
                 * ptr->usym.vsymt.vtypep->usym.asymt.size) );
  }  /* fixrsize */

/* ------------------------------------------------------------------ */

 symtabp enterrange(upper,lower,sy)
  int upper,lower;
  symtype sy;
   /* Enter a subrange node into symbol table, but do not link into hash
      chain */

  { symtabp sptr;

    if (sy == ARRAY_TYPE) sptr = nohash(ARRAY_TYPE, "<array>");
       else sptr = nohash(SUBRANGE, "<subrange>");
    sptr->usym.asymt.udim = upper;
    sptr->usym.asymt.ldim = lower;
    sptr->usym.asymt.subtype = NULL;
    sptr->usym.asymt.size = upper - lower + 1;
    return(sptr);
  }  /* enterrange */
```

```
/* -------------------------------------------------------------------- */

 enterproc(sptr)
  symtabp sptr;
   /* Fills in values for process node in the symbol table. */

  { if (sptr->symt != PROC_TYPE)
      { sptr->symt = PROC_TYPE;
        sptr->usym.psymt.procnum = nxtproc++;
        sptr->usym.psymt.dim = -1;
        sptr->usym.psymt.parproc = NULL;
      }
    sptr->rlevel = rlevel;
    sptr->usym.psymt.nxtproc = sptr ->usym.psymt.labptr = NULL;
    sptr->usym.psymt.noprocs = 1;  /* assume only one instance of process */
    sptr->usym.psymt.locals = 1;   /* no local vars;
                                      reserve space for return address */
    sptr->usym.psymt.defined = FALSE;
   }  /* enterproc */

/* -------------------------------------------------------------------- */

 boolean prochier(new, old)
  symtabp new, old;  /* where new is the process to communicate with and
                        old is the process currently being compiled */
   /* Checks whether a process may interface with another without
      deadlock; for example when the one process is nested within
      the other. Because multi-defined processes are allowed, no check
      made to ensure that process is not communicating with itself. */

  { if (new == old)  /* If multi-instance process return false */
       if (!old->usym.psymt.dim)  return(TRUE);
           else return(FALSE);
     else if (old == NULL) return(FALSE);
            else return( prochier (new, old->usym.psymt.parproc) );
  }  /* prochier */

/* -------------------------------------------------------------------- */

 symtabp enter_comp(sptr, dim, size, offset)
  int *dim, offset, *size;
  semrecp sptr;
   /* Enter VARI nodes for the components of a message var */

   {  symtabp yptr;
      register int k;

      if (sptr == NULL) return(NULL);
        else
           { yptr = nohash(VARI, "<messitem>");
             (*dim)++;
             yptr->usym.vsymt.saddr = offset;
             yptr->usym.vsymt.canchange = TRUE;
             yptr->usym.vsymt.vtypep = sptr->usem.insemt.idsymp;
             k = type_size(yptr->usym.vsymt.vtypep);
             *size += k;
             yptr->usym.vsymt.nxtvar =
                 enter_comp(sptr->usem.insemt.idnext, dim, size, offset+k);
```

```
            return(yptr);
          }
   }  /* enter_comp */

/* ---------------------------------------------------------------------- */

 symtabp set_bintype(tag,left,right)
  semtype tag;
  symtabp left, right;
   /* Called on a binary operator (tag), this verifies that both operands
      (left, right) are compatible with the operator, then returns a result
      type -- a symbol table pointer to one of the three primitive types */

  { if ( check_type(tag,left->symt) && check_type(tag,right->symt) )
      { if (math(tag))
            return ( (utype(left->symt,right->symt) == REAL_TYPE) ? realptr
                     : intptr );
          else if (boo(tag)) return(boolptr);
                else { error("Binary type screwup"); return(intptr); }
       }
     else return(intptr);
  }  /* set_bintype */

/* ---------------------------------------------------------------------- */

 semrecp gen_binnode(tag)
  semtype tag;
   /* Creates and returns a new binary operator node, covering the two
      operand nodes.  Operands are checked for compatibility with operator.
      If operands can be subsumed as constants, that is done, returning
      a constant. */

  { semrecp sp;

    sp = newsem();
    sp->semt = tag;
    sp->usem.msemt.treel = semstack[stackx-2];
    sp->usem.msemt.treer = semstack[stackx];
    sp->svtype = set_bintype(tag, sp->usem.msemt.treel->svtype,
                             sp->usem.msemt.treer->svtype);
    if (is_const(sp->usem.msemt.treel->semt) &&
         is_const(sp->usem.msemt.treer->semt) && arithtypes(tag) )
           cbin_arith(tag,sp->svtype,sp->usem.msemt.treel,
                      sp->usem.msemt.treer,sp);
    return(sp);
  }  /* gen_binnode */

/* ---------------------------------------------------------------------- */

 semrecp gen_unary(tag)
  semtype tag;
   /* Creates a new unary operator node pointing to an operand.
      Checks operand for compatibility with operator.  If operand is a
      constant, constant operation is performed, returning constant node. */

  { semrecp t1semp, t2semp;

    t1semp = semstack[stackx];
```

```
   if (check_type(tag,tlsemp->svtype->symt))
      { if ( is_const(tlsemp->semt) && ( (tag == NEG_NODE) ||
             (tag == NOT_NODE) ) )
          { un_arith(tag, tlsemp);
            return(tlsemp);  /* semstack[stackx] must be preserved */
          }
         else { t2semp = newsem();
                t2semp->semt = tag;
                t2semp->usem.treeun = tlsemp; /* points to semstack */
                t2semp->svtype = tlsemp->svtype;
                return(t2semp);
              }
      }
    else return(tlsemp);
 }  /* gen_unary */

/* ---------------------------------------------------------------------- */

semrecp gen_relnode(tag)
 semtype tag;
  /* This just prepares a node for subsequent augmentation by GEN_BINNODE */

 { semrecp tlsemp;

   tlsemp = newsem();
   tlsemp->svtype = boolptr;
   tlsemp->semt = tag;
   tlsemp->usem.msemt.treel = NULL;
   tlsemp->usem.msemt.treer = NULL;
   return(tlsemp);
 }  /* gen_relnode */

/* ---------------------------------------------------------------------- */

semrecp bad_var(msg,ptr)
 char msg[];
 symtabp ptr;
  /* Called when some invalid symbol is found that is supposed to
     be a variable or process.  This makes one so that semantics
     operations can continue. */

 { semrecp sptr;

   sptr = newsem();
   sptr->semt = IDENT;
   sptr->svtype = intptr;
   sptr->usem.idsemt.vext = NULL;
   if (ptr == NULL)
      { error(msg);
        ptr = makesym(errsym, VARI, plevel);
        ptr->usym.vsymt.saddr = 0;
        ptr->usym.vsymt.vtypep = intptr;
        ptr->usym.vsymt.canchange = TRUE;
        ptr->usym.vsymt.nxtvar = NULL;
      }
    else { warning(msg);
           enterproc(ptr);
           ptr->level = 0;  /* level = global */ }
```

```
   sptr->usem.idsemt.symp = ptr;
   return(sptr);
 }  /* bad_var */

/* ---------------------------------------------------------------------- */

 semrecp make_arnext(bptr,semt,dim)
  semrecp bptr;
  int *dim;
  semtype semt;
   /* This works on a list of expressions intended as array indexes.
      It converts a BLIST_NODE  list into a list of AR_EXT forms. */

  { semrecp temp;

    if (bptr == NULL) return(NULL);
     else { (*dim)++;
            temp = newsem();
            temp->svtype = NULL; /* set later */
            temp->semt = semt;
            temp->usem.arsemt.arsymp = NULL;  /* also set later */
            temp->usem.arsemt.ax = bptr->usem.bsemt.blexp;
            temp->usem.arsemt.arextnext =
               make_arnext(bptr->usem.bsemt.blnext, semt, dim);
            return(temp);
          }
   }  /* make_arnext */

/* ---------------------------------------------------------------------- */

 static symtabp type_of(semp,tb)
  semrecp semp;
  symtabp tb;
   /* Recursively does most of the work for type_of_var */

  { symtabp tsymp;

     if (tb == NULL)
            error("Invalid array name -- too many dimensions");
      else if (semp == NULL)  /* successfully scanned extensions */
             return ( (tb->symt == SUBRANGE) ? intptr : tb);
       else if ( (semp->semt == AR_EXT) && (tb->symt == ARRAY_TYPE) )
              { semp->usem.arsemt.arsymp = tb;
                return(type_of(semp->usem.arsemt.arextnext,
                                tb->usym.asymt.subtype));
              }
         else if (semp->semt != SEM_ERR)
                 error(" Invalid compound identifier");
     return(NULL); /* default type */
  }  /* type_of */

/* ---------------------------------------------------------------------- */

 static symtabp proc_type_of(semp,tb)
  semrecp semp;
  symtabp tb;
   /* Recursively does most of the work for type_of_var */
```

```
{ symtabp tsymp;

   if ((tb == NULL) && (semp == NULL))
        return(intptr); /* successfully scanned extensions */
     else if ( (semp->semt == PROC_EXT) && (tb->symt == VARI) )
          { semp->usem.arsemt.arsymp = tb;
            return(proc_type_of(semp->usem.arsemt.arextnext,
                                tb->usym.vsymt.nxtvar));
          }
       else if (semp->semt != SEM_ERR)
                error(" Process label required or invalid");
    return(NULL); /* default type */
  } /* proc_type_of */

/* ---------------------------------------------------------------------- */

symtabp type_of_var(semp)
 semrecp semp;
  /* Establishes type of a compound variable.  Variable could terminate
     early, returning a non-primitive type. This is OK for addressing
     purposes.  Otherwise it also checks for consistency between the
     compound name components and what has been declared for the variable's
     type structure. It sets the ARSYMP field for an AR_EXT node. */

  {  if (semp != NULL)
      if (semp->semt == IDENT)
        { if (semp->usem.idsemt.symp->symt == VARI)
            return( type_of(semp->usem.idsemt.vext,
                            semp->usem.idsemt.symp->usym.vsymt.vtypep) );
           else if (semp->usem.idsemt.symp->symt == PROC_TYPE)
                 return( (semp->usem.idsemt.symp->usym.psymt.defined) ?
                         proc_type_of(semp->usem.idsemt.vext,
                                  semp->usem.idsemt.symp->usym.psymt.labptr) :
                         intptr);
            else error("Type of var disaster 1");
        }
       else if (semp->semt != SEM_ERR) error("Type of var disaster 2");
     return(NULL);
  }  /* type_of_var */

/* ---------------------------------------------------------------------- */

semrecp make_consnode(sp)
 symtabp sp;
  /* Create a new semrec node from a constant identifier */

 { semrecp semp;

   semp = newsem();
   switch (sp->usym.csymt.constyp)
      { case FIXED: semp->semt = FIXED; semp->svtype = intptr;
                    semp->usem.numval = sp->usym.csymt.consival;
                    break;
        case FLT: semp->semt = FLT; semp->svtype = realptr;
                  semp->usem.rval = sp->usym.csymt.consrval;
                  break;
        case BOOLS: semp->semt = BOOLS; semp->svtype = boolptr;
                    semp->usem.numval = sp->usym.csymt.consival;
```

```
                   break;
         case STRNG: semp->semt = STRNG; semp->svtype = NULL;
                     semp->usem.stx = sp->usym.csymt.consival;
                     break;
      }
    return(semp);
  }  /* make_consnode */

/* ---------------------------------------------------------------------- */

 static boolean bmatch(tarbl, expbl)
  semrecp tarbl, expbl;     /* blist node pointers */
   /* Checks for compatibility between messages with an empty constructor */

  {  if (!tarbl && !expbl) return(TRUE);
     if (!tarbl || !expbl) return(FALSE);
     if (tarbl->usem.bsemt.blexp->svtype != expbl->usem.bsemt.blexp->svtype
         && !check_compat(tarbl->usem.bsemt.blexp->svtype,
                          expbl->usem.bsemt.blexp->svtype, FALSE) )
        return(FALSE);
     return(bmatch(tarbl->usem.bsemt.blnext, expbl->usem.bsemt.blnext));
  }  /* bmatch */

/* ---------------------------------------------------------------------- */

 int struct_assign(target, exp)
  semrecp target, exp;
   /* Depending on whether target and exp are structured values, decides
      what form of assignment should take place */

  { if (target->semt == MESS_NODE)
       if (exp->semt == MESS_NODE)
         { if ( !(target->usem.gsemt.messvar && exp->usem.gsemt.messvar)
                && !bmatch(target->usem.gsemt.messexp,
                           exp->usem.gsemt.messexp) )
                /* checks messages with no constructor first */
                  error("Messages not compatible");
             else if (target->usem.gsemt.messvar->usym.asymt.udim !=
                      exp->usem.gsemt.messvar->usym.asymt.udim )
                error("Messages not compatible"); /* not same constructor */
           return(1);
          }
         else error("Expression not a structured value");
       else if (target->semt == SIG_NODE)
              if (exp->semt == SIG_NODE) return(2);
                else error("Expression should be a signal");
              else if ( (exp->semt == SIG_NODE) || (exp->semt == MESS_NODE) )
                     error("Target not a structured variable");
    if (target->usem.idsemt.symp->symt != VARI)
           error("Invalid target variable");
    return(0);
  }  /* struct_assign */

/* ---------------------------------------------------------------------- */

 boolean check_mvar(tl,b)
  symtabp tl;
  semrecp b;
```

```
  /* Check if use of a message is compatible with its declaration */

 { symtabp t2;

   t2 = b->usem.bsemt.blexp->svtype;

   while (t1 != NULL)
        { if ( (t2 == NULL) || ( (t1->usym.vsymt.vtypep != t2) &&
             (!check_compat(t1->usym.vsymt.vtypep,t2,FALSE)) ))
                return(FALSE);
          t1 = t1->usym.vsymt.nxtvar;
          b = b->usem.bsemt.blnext;
          if (b != NULL) t2 = b->usem.bsemt.blexp->svtype;
        }
    if (b != NULL) return(FALSE);
    return(TRUE);
  }  /* check_mvar */

/* ------------------------------------------------------------------------ */

multirange(sptr,jumpadr,vals,dims)
 int jumpadr, vals[], dims;
 semrecp sptr; /* blist node */
  /* Generates jump table entries for labelled guards - called by
     jumprange() */

  { symtabp srg;
    int lrang, urang;
    register int i;

    if (sptr == NULL)
       { gen_3byte(AJP,jumpadr,dims);
         for (i = dims-1; i >= 0; i--) genbyte(vals[i]);
         emitnl();
         return;
       }
    srg = sptr->usem.bsemt.blexp->usem.idsemt.symp->usym.vsymt.vtypep;
    lrang = srg->usym.asymt.ldim;
    urang = srg->usym.asymt.udim;
    if (dims == MAXDIM) error("Guard range has too many labels");
     else for (i=lrang; i<=urang; i++)
            { vals[dims] = i;
              multirange(sptr->usem.bsemt.blnext,jumpadr,vals,dims+1);
            }
  }  /* multirange */

/* ------------------------------------------------------------------------ */

jumprange(lext, adr)
 semrecp lext;  /* blist node */
 int adr;
  /* Calls multirange to generate jump table entries */

  { int vals[MAXDIM-1];
    multirange(lext,adr,vals,0);
  }  /* jumprange */
```

```
/* ------------------------------------------------------------------------ */

 chkdef()
  /* Reports any undefined processes at the global level */

  { register int hx;
    symtabp sp;
    boolean uns = FALSE;

    for (hx = 0; hx <= HLIMIT; hx++)
     { sp = symtab[hx];
       while (sp != NULL)
         { if ( sp->symt == PROC_TYPE && !(sp->usym.psymt.defined) )
             { if (!uns) fprintf(rfile,"Error: processes undefined -> ");
               uns = TRUE;
               fprintf(rfile, "%s  ", sp->sym);
             }
           sp = sp->next;
         }
      }
    if (uns || gencode < 0)
         { fputc('\n',rfile);
           fprintf(rfile, "\nCompilation errors!\n"); }
      else if (!gencode) fprintf(rfile, "\n\nCompiled correctly!\n");
  }  /* chkdef */

/* ------------------------------------------------------------------------ */
```

```
 /* Code generation routines for the CSP-i parser */

#include "CSPDEFS.C"
#include "CSPALL.C"
#include "CSPCODE.C"

 extern symtype utype();
 extern boolean is_std();

/* ----------------------------------------------------------------------- */

 init_sem()
  /* Semantics initialization -- called before any productions are applied.
     Enters predefined types into symbol table. */

  { boolptr = makesym("BOOLEAN",BOOL_TYPE, -1);
    boolptr->usym.asymt.subtype = NULL;
    boolptr->usym.asymt.size = BOOL_SIZE;
    charptr = makesym("CHAR",CHAR_TYPE, -1);
    charptr->usym.asymt.subtype = NULL;
    charptr->usym.asymt.size = CHAR_SIZE;
    intptr = makesym("INTEGER",INT_TYPE, -1);
    intptr->usym.asymt.size = INT_SIZE;
    intptr->usym.asymt.subtype = NULL;
    realptr = makesym("REAL",REAL_TYPE, -1);
    realptr->usym.asymt.size = REAL_SIZE;
    realptr->usym.asymt.subtype = NULL;
  }  /* init_sem */

/* ----------------------------------------------------------------------- */

 end_sem()
  /* Semantics conclusion -- called after the GOAL production is applied. */
  { errpos = -2;
    if (gencode > 0) error("GENCODE disaster");
    gen_2byte(HLT,-1);     /* halt entire program */
    chkdef();  /* find any undefined functions */
  }  /* end_sem */

/* ----------------------------------------------------------------------- */

 dumps(ch)
  char ch;
   /* Emit opcodes for stack and process descriptor dumps */

  { if (ch == PRDUMP) gen_1byte(DMP);
     else gen_1byte(STK);
  }  /* dumps */

/* ----------------------------------------------------------------------- */

 int wrtype(typ,relev)
  symtype typ;
  boolean relev;  /* important to differentiate between char and int */
   /* Writes I, A or R to an instruction based on typ */

  { if (relev && typ == CHAR_TYPE) return(CHR);
     else if ( (typ == INT_TYPE) || (typ == BOOL_TYPE) || (typ == CHAR_TYPE))
```

```
                return( ITG);
      else if (typ == REAL_TYPE) return(REL);
        else if (typ == ARRAY_TYPE) return( ARR);
          else symerror(symtypename[typ],"WRTYPE disaster");
  }  /* wrtype */

/* ---------------------------------------------------------------------------- */

 emitnl()
  /* Writes a newline to codefile */

  { if (!gencode) fputc('\n',cfile); }

/* ---------------------------------------------------------------------------- */

 genbyte(byt)
  short int byt;
   /* Writes a byte to the codefile */

  {  if (!gencode) fprintf(cfile, " %d", byt); }

/* ---------------------------------------------------------------------------- */

 gen_1byte(inst)
  short int inst;
   /* Writes a byte to the codefile, followed by EOLN */

  { if (!gencode)
      if (inst == MSI || inst == MSO) fprintf(cfile, "%d", inst);
       else fprintf(cfile, "%d\n", inst);
  }  /* gen_1byte */

/* ---------------------------------------------------------------------------- */

 gen_real(inst, no)
  short int inst;
  float no;
   /* Writes an inst plus one real attribute to target file */

  { if (!gencode) fprintf(cfile, "%d %f\n" , inst, no); }

/* ---------------------------------------------------------------------------- */

 gen_2byte(inst, no)
  short int inst;
  int no;
   /* Writes an inst plus one attribute to target file */

  { if (!gencode)
       if (inst == LDS) fprintf(cfile, "%d %d" , inst, no);
         else fprintf(cfile, "%d %d\n" , inst, no);
  }  /* gen_2byte */

/* ---------------------------------------------------------------------------- */

 gen_3byte(inst, no, val)
  int no, val;
  short int inst;
```

```
 { if (!gencode)
        if (inst == AJP) fprintf(cfile, "%d %d %d" , inst, no, val);
         else fprintf(cfile, "%d %d %d\n" , inst, no, val);
 }  /* gen_3byte */

/* ------------------------------------------------------------------------ */

gen_4byte( inst, no, a, b)
 int no,a,b;
 short int inst;
  /* Writes an inst plus three attributes to target file */

 { if (!gencode) fprintf(cfile, "%d %d %d %d\n" , inst, no,a,b); }

/* ------------------------------------------------------------------------ */

gen_5byte( inst, no, a, b,c)
 int no,a,b,c;
 short int inst;
  /* Writes an inst plus four attributes to target file */

  { if (!gencode) fprintf(cfile, "%d %d %d %d %d\n" , inst, no,a,b,c); }

/* ------------------------------------------------------------------------ */

gen_inst( inst,tp)
 short int inst;
 symtype tp;
  /* Writes an instruction to code file */

 { gen_lbyte( inst+wrtype(tp,FALSE)); }

/* ------------------------------------------------------------------------ */

gen_ninst( inst, tp)
 short int inst;
 symtype tp;
  /* Writes an instruction as a INST/NOT pair */

 {  gen_inst( inst,tp); gen_lbyte( NOT); }

/* ------------------------------------------------------------------------ */

conv_type( tl, t2)
 symtype tl, t2;
  /* Writes FLO or FIX to code file to convert type 1 to type 2 */

{ if ( is_std(tl) && is_std(t2) && (tl != t2) )
     { if (tl == REAL_TYPE) gen_lbyte( FIX);
          else if (t2 == REAL_TYPE) gen_lbyte( FLO);
       if (tl == BOOL_TYPE || t2 == BOOL_TYPE)
                 error( "BOOLEAN type required");
     }
 }  /* conv_type */
```

```
/* ---------------------------------------------------------------------- */

 gen_plab(typtr)
  symtabp typtr;
   /* Writes a label statement to target file. The label parameters
      are the lower and upper dimensions of a subrange node */

  { if (!gencode)
      gen_3byte(LAB, typtr->usym.asymt.ldim,typtr->usym.asymt.udim);
  }  /* gen_plab */

/* ---------------------------------------------------------------------- */

 convert (semp,tp)
  semrecp semp;
  symtype tp;
   /* Mutates semp by constant conversion, so that a REAL constant
      may be truncated, an INTEGER or CHAR may be floated */

  { if (semp == NULL) error ("Convert disaster");
      else if (semp->svtype->symt != tp)
            { /* do something only if types differ */
              if (tp == REAL_TYPE)
                 { semp->usem.rval = semp->usem.numval;
                   semp->semt = FLT;
                   semp->svtype = realptr;
                 }
                else if (semp->svtype->symt == REAL_TYPE)
                       { semp->usem.numval = semp->usem.rval;
                         semp->svtype = intptr;
                         semp->semt = FIXED;
                       }
                 else if ( !(tp == INT_TYPE || tp == CHAR_TYPE) ||
                           !(semp->svtype->symt == INT_TYPE ||
                             semp->svtype->symt == CHAR_TYPE) )
                      error ("Incompatible types");
            }
   }  /* convert */

/* ---------------------------------------------------------------------- */

 un_arith(tag,semp)
  semtype tag;
  semrecp semp;
   /* Performs NOT and NEG for constant child */

  { if (semp == NULL) error("Un_arith disaster");
     else /* mutate semp in place */
      if (tag == NEG_NODE)
         if (semp->semt == FIXED) semp->usem.numval = - semp->usem.numval;
            else semp->usem.rval = - semp->usem.rval;
       else /* tag == NOT_NODE */
         { semp->semt = BOOLS;
           semp->usem.numval = ! semp->usem.numval; }
  }  /* un_arith */
```

```
/* ------------------------------------------------------------------------ */

 cbin_arith(tag,sp,left,right,semp)
  semtype tag;
  symtabp sp;
  semrecp left, right, semp;
   /* This performs compile-time evaluation of constant binary operator
      expressions.  Constant folding is performed during the bottom-up
      construction of an expression tree, and is triggered by finding a
      unary or binary operator with constant children. */

   { boolean rl;
     symtype ytag;
     float rrval;
     int nnval;

     rl = FALSE; /* assume numval is changed */
     if (semp == NULL) error("Cbin_arith disaster");
      else
       { semp->svtype = sp;
         switch (sp->symt)
          { case BOOL_TYPE: semp->semt = BOOLS; break;
            case CHAR_TYPE:
            case INT_TYPE : semp->semt = FIXED; break;
            case REAL_TYPE : semp->semt = FLT; break;
          }
         switch (tag)
          {
            case OR_NODE: nnval = left->usem.numval || right->usem.numval;
                          break;
            case AND_NODE: nnval = left->usem.numval && right->usem.numval;
                           break;
            case EQ_NODE:
            case GEQ_NODE:
            case GTR_NODE:
            case LEQ_NODE:
            case LES_NODE:
            case NEQ_NODE: /* here sp is a boolean type */
              ytag = utype(left->svtype->symt,right->svtype->symt);
              convert(left,ytag); convert(right,ytag);
              if (ytag == INT_TYPE || ytag == CHAR_TYPE)
              switch (tag)
               { case EQ_NODE:
                   nnval = left->usem.numval == right->usem.numval; break;
                 case GEQ_NODE:
                   nnval = left->usem.numval >= right->usem.numval; break;
                 case GTR_NODE:
                   nnval = left->usem.numval > right->usem.numval; break;
                 case LEQ_NODE:
                   nnval = left->usem.numval <= right->usem.numval; break;
                 case LES_NODE:
                   nnval = left->usem.numval < right->usem.numval; break;
                 case NEQ_NODE:
                   nnval = left->usem.numval != right->usem.numval; break;
               }
              else /* ytag == REAL_TYPE */
               switch (tag)
                { case EQ_NODE:
```

```
          nnval = left->usem.rval == right->usem.rval; break;
        case GEQ_NODE:
          nnval = left->usem.rval >= right->usem.rval; break;
        case GTR_NODE:
          nnval = left->usem.rval > right->usem.rval; break;
        case LEQ_NODE:
          nnval = left->usem.rval <= right->usem.rval; break;
        case LES_NODE:
          nnval = left->usem.rval < right->usem.rval; break;
        case NEQ_NODE:
          nnval = left->usem.rval != right->usem.rval; break;
      }
    break;

   case ADD_NODE:
   case SUB_NODE:
   case MOD_NODE:
   case MPY_NODE:
   case QUOT_NODE:
      convert(left,sp->symt); convert(right,sp->symt);
      if (sp->symt == INT_TYPE || sp->symt == CHAR_TYPE)
      switch(tag)
        { case ADD_NODE:
            nnval = left->usem.numval + right->usem.numval; break;
          case SUB_NODE:
            nnval = left->usem.numval - right->usem.numval; break;
          case MOD_NODE:
             if (right->usem.numval <= 0)
                  { error(" Invalid divisor"); nnval = 0; }
                else nnval = left->usem.numval % right->usem.numval;
             break;
          case MPY_NODE:
            nnval = left->usem.numval * right->usem.numval; break;
          case QUOT_NODE:
            if (right->usem.numval == 0)
                  { error(" Zero divisor"); nnval = 0; }
              else nnval = left->usem.numval / right->usem.numval;
            break;
        }
      else /* symt == REAL_TYPE */
       { rl = TRUE;
         switch(tag)
           { case ADD_NODE:
               rrval = left->usem.rval + right->usem.rval; break;
             case SUB_NODE:
               rrval = left->usem.rval - right->usem.rval; break;
             case MOD_NODE:
               rrval = 1.0; error("MOD illegal with reals"); break;
             case MPY_NODE:
               rrval = left->usem.rval * right->usem.rval; break;
             case QUOT_NODE:
               if (right->usem.numval == 0)
                     { error(" Zero divisor"); rrval = 0; }
                else rrval = left->usem.rval / right->usem.rval;
                break;
           }
       } /* else */
      break;
```

```
        } /* main tag switch */
    if  (rl == TRUE) semp->usem.rval = rrval;
       else semp->usem.numval = nnval;
   } /* else */
 }  /* cbin_arith */

/* ----------------------------------------------------------------------- */

bin_arith(st, left, right)
 semrecp left, right;
 semtype st;
  /* Evaluates left and right members of a binary arithmetic node,
     then writes a postfix operator to the code file */

 { symtype rt;  /* type to which each operand must first be converted */

   rt = utype( left->svtype->symt, right->svtype->symt);
   eval( left, rt, NULL);  /* left operand evaluation to type rt */
   eval( right, rt, NULL); /* right operand evaluation to type rt */
   switch (st)
    { case ADD_NODE: gen_inst( ADI,rt); break;
      case SUB_NODE: gen_inst( SBI,rt); break;
      case MPY_NODE: gen_inst( MLI,rt); break;
      case QUOT_NODE: gen_inst( DVI,rt); break;
      case MOD_NODE: if (rt != INT_TYPE && rt != CHAR_TYPE)
                           error( "MOD only supported for integer types");
                         else gen_lbyte( MOD);
                       break;
      case NEQ_NODE: gen_ninst( EQI,rt); break;
      case EQ_NODE: gen_inst( EQI,rt); break;
      case LEQ_NODE: gen_ninst( GTI,rt); break;
      case GTR_NODE: gen_inst( GTI,rt); break;
      case GEQ_NODE: gen_ninst( LSI,rt); break;
      case LES_NODE: gen_inst( LSI,rt); break;
    }
 }  /* bin_arith */

/* ----------------------------------------------------------------------- */

bin_logic(st, left, right)
 semtype st;
 semrecp left, right;
  /* Similar to EVAL, but for boolean evaluation */

 {  eval (left, BOOL_TYPE, NULL);
    eval (right, BOOL_TYPE, NULL);
    if (st == AND_NODE) gen_lbyte( AND);
      else gen_lbyte( ORR);
 }  /* bin_logic */

/* ----------------------------------------------------------------------- */

semrecp gen_offset(semp, offset)
 semrecp semp;
 int *offset;
  /* Walks through the SEMP chain adjusting the OFFSET for constant array
     indexing and process labels.  Stops on a SEMP that represents an
     array or label run-time index.  Returns the adjusted OFFSET
```

```
    so far (may be unchanged) and the offending SEMP (may be same as
    passed to it. */

 { int inx, k;
   symtabp symp,symp2;

  while (semp != NULL)
    if (semp->semt == AR_EXT)
       { symp = semp->usem.arsemt.arsymp;
         if ( is_const(semp->usem.arsemt.ax->semt)) /* constant index */
          { if (semp->usem.arsemt.ax->semt == FLT)
              { error("Floating point index");
                inx = semp->usem.arsemt.ax->usem.rval;
              }
            else inx = semp->usem.arsemt.ax->usem.numval;
            if (inx < symp->usym.asymt.ldim || inx > symp->usym.asymt.udim)
               error ("Index out of range");
            *offset = *offset + (inx - symp->usym.asymt.ldim) *
                           symp->usym.asymt.subtype->usym.asymt.size;
            semp = semp->usem.arsemt.arextnext;
          }
         else /* index not constant */ return(semp);
        }
     else if (semp->semt == PROC_EXT)
       { symp = semp->usem.arsemt.arsymp->usym.vsymt.vtypep;
         symp2 = semp->usem.arsemt.arsymp->usym.vsymt.nxtvar;
         if ( is_const(semp->usem.arsemt.ax->semt)) /* constant index */
          { if (semp->usem.arsemt.ax->semt != FIXED)
              { error("Floating point or boolean label");
                inx = (semp->usem.arsemt.ax->semt == FLT) ?
                       semp->usem.arsemt.ax->usem.rval :
                       semp->usem.arsemt.ax->usem.numval;
              }
           else inx = semp->usem.arsemt.ax->usem.numval;
           k =  (symp2 == NULL) ? 1
                          : symp2->usym.vsymt.vtypep->usym.asymt.size;
           if (inx < symp->usym.asymt.ldim || inx > symp->usym.asymt.udim)
               error ("Label out of range");
           *offset = *offset + (inx - symp->usym.asymt.ldim) * k;
           semp = semp->usem.arsemt.arextnext;
          }
         else /* index not constant */ return(semp);
       }
      else if (semp->semt == MES_EXT)
             { *offset = *offset + semp->usem.offset;
               return(NULL);
             }
          else  { error("GENOFFSET disaster"); break; }
  return(NULL);
 } /* gen_offset */

/* ----------------------------------------------------------------------- */

gen_var(semp, inst)
 int inst;
 semrecp semp;
  /* Generates optimized code for a variable rooted in SEMP. INST must
    be one of LOAD, STOR or LOAD @. */
```

```
{ int offset,   /* base address */
      storsize, /* number of bytes to store if array type */
      lev, k;
  symtype vtype;
  symtabp symp;

  if (!gencode)
    { offset = semp->usem.idsemt.symp->usym.vsymt.saddr;
      vtype = semp->svtype->symt; /* variable type */
      storsize = semp->svtype->usym.asymt.size;
      lev = rlevel - semp->usem.idsemt.symp->rlevel; /* relative level */
      semp = gen_offset(semp->usem.idsemt.vext, &offset);
          /* offset gets changed */
      if (semp == NULL)
        { /* a complete compile-time offset could be computed --
              apparently no variable array indices */
          if (inst == LID) gen_3byte(inst,lev,offset);
            else  ((k = wrtype(vtype,FALSE)) == ARR) ?
                 gen_4byte(inst+k,storsize,lev,offset) :
                 gen_3byte(inst+k,lev,offset);
        }
      else
        /* the general case -- we've hit an array with a variable address,
            so have to compute an address first, then adjust it */
        { gen_3byte(LID, lev, offset);
          /* the initial address, with the offset computed so far */
          do { /* compound variable; semp != NULL */
             if (semp->semt == AR_EXT)
               { /* array index that is run-time computable */
                 eval(semp->usem.arsemt.ax, INT_TYPE, NULL);
                 symp = semp->usem.arsemt.arsymp;
                 gen_4byte(IND, symp->usym.asymt.ldim,
                           symp->usym.asymt.udim,
                           symp->usym.asymt.subtype->usym.asymt.size);
               }
              else error("GEN_VAR disaster");

              offset = 0; /* start next offset */
              semp = gen_offset(semp->usem.arsemt.arextnext, &offset);
              if (offset > 0)
                 { gen_2byte(PSI, offset);
                   gen_1byte(ADI); }
             }
           while (semp != NULL);

         if (inst != LID) /* LID is done */
            { if ( (k=wrtype(vtype,FALSE)) == ARR)
                 gen_2byte(inst+k+STLD, storsize);
               else gen_1byte(inst+k+STLD);
             }
        } /* else general case */
    }
} /* gen_var */
```

```
/* ------------------------------------------------------------------------ */

 eval_addr(semp)
  semrecp semp;
   /* Yields a stack address of an object SEMP.  This works for
      partially-specified compound objects, for simple primitive variables
      and for compound primitive variables. */

   { if (semp == NULL) error("EVAL_ADDR disaster 3");
       else if ( (semp->semt == IDENT) &&
                 (semp->usem.idsemt.symp->symt == VARI) )
                    gen_var(semp, LID);
           else error("invalid address form");
   }  /* eval_addr */

/* ------------------------------------------------------------------------ */

 int check_compat(sor, dest, report)
  symtabp sor, dest;
  boolean report;
   /* Check the compatibility of array assignments */

  { if ( (dest->symt == ARRAY_TYPE)
          && (dest->usym.asymt.size == sor->usym.asymt.size) )
        return(dest->usym.asymt.size);
    if (report) error("Incompatible array types");
    return(0);
  }  /* check_compat */

/* ------------------------------------------------------------------------ */

 eval(sp, totype, typtr)
  semrecp sp;
  symtype totype;
  symtabp typtr;
   /* Master AST evaluator.  This takes a subset of the SEMREC structures
      and generates evaluation code for the whole structure. The end result
      is also converted to the type TOTYPE. Some SEMREC types have no
      meaning in EVAL and result in a complaint -- see the case tags for
      those that can be processed. */

  {  semrecp ln, rn;
     symtype rt;
     register int i;

     if ( (!gencode) && (sp != NULL) )
      { if (sp->svtype == NULL) rt = UNK_TYPE;
         else rt = sp->svtype->symt; /* the default result type */

        switch (sp->semt)
         { case ADD_NODE:
           case SUB_NODE:
           case MPY_NODE:
           case QUOT_NODE:
           case MOD_NODE:
           case EQ_NODE:
           case NEQ_NODE:
           case GTR_NODE:
```

```
          case LES_NODE:
          case GEQ_NODE:
          case LEQ_NODE:
             bin_arith(sp->semt,sp->usem.msemt.treel,sp->usem.msemt.treer);
             break;
          case AND_NODE:
          case OR_NODE:
             bin_logic(sp->semt,sp->usem.msemt.treel,sp->usem.msemt.treer);
             break;
          case NEG_NODE:
             eval(sp->usem.treeun,sp->svtype->symt,sp->svtype);
             gen_1byte(NGI+wrtype(sp->svtype->symt,FALSE));
             break;
          case NOT_NODE:
             eval(sp->usem.treeun, BOOL_TYPE, NULL);
             gen_1byte(NOT); break;
          case IDENT:
             if ( is_std(rt)) gen_var(sp,LDI);
               else if (rt == ARRAY_TYPE)
                       { gen_var(sp, LID);
                         gen_2byte(LOA,
                         check_compat(sp->svtype, typtr, TRUE) );
                         rt = totype; }
                     else symerror(sp->usem.idsemt.symp->sym,
                                   " - invalid identifier type ");
             break;
          case FIXED:
          case FLT:
          case BOOLS:
             convert(sp, totype);
             if (!gencode)
                 if (sp->semt == FLT) gen_real(PSR, sp->usem.rval);
                     else gen_2byte(PSI, sp->usem.numval);
             rt = totype; break;
          case STRNG:
             gen_2byte(LDS, strlen(&strtab[sp->usem.stx]) );
             for (i=0;strtab[sp->usem.stx+i] != '\n'; i++)
                  genbyte(strtab[sp->usem.stx+i]);
             emitnl();
             if (totype != ARRAY_TYPE) error("String type invalid");
             rt = totype; break;
          case SIG_NODE:  /* following used only in error cases */
          case MESS_NODE:
              error("Invalid use of a structured variable");
              gen_1byte("LOAD SIG"); rt = totype;
              break;
          default:
              error("EVAL disaster 1"); rt = totype; break;
        } /* switch */
      conv_type(rt, totype);
    }
  } /* eval */

/* ---------------------------------------------------------------------- */

 gen_assign(targ,semexpr)
  semrecp targ, semexpr;
   /* This generates code for the assignment statement.  TARG may be
```

```
      simple or compound */

   { if (!gencode)
      { eval(semexpr, targ->svtype->symt, targ->svtype);
        /* generate code for expression, converting to type of
           receiving variable */
        eval_stor(targ, STI);  /* then store result */
       }
   }  /* gen_assign */

/* ------------------------------------------------------------------------ */

 fix_messvar (sp)
  semrecp sp;
   /* Inserts a MES_EXT node to enable gen_offset to adjust the
      variable offset accordingly */

  { semrecp sptr;

      sptr = newsem();
      sptr->semt = MES_EXT;
      sptr->svtype = NULL;
      sp->usem.idsemt.vext = sptr;
  }  /* fix_messvar */

/* ------------------------------------------------------------------------ */

 gen_sassign(ids, bexps)
  semrecp bexps, ids;
   /* This generates code for a structured assignment statement */

  { boolean mexp, mtarg;
    symtabp t1, t2;
    semrecp a,b;

     if ( (ids == NULL) || (bexps == NULL) ) return;
     if (mtarg = (ids->semt != BLIST_NODE))
          {t1 = ids->svtype->usym.asymt.subtype;
           fix_messvar(ids);
          }
     if (mexp = (bexps->semt != BLIST_NODE))
          { t2 = bexps->svtype->usym.asymt.subtype;
            fix_messvar(bexps);
          }
     while (TRUE)
       { if (mtarg)
            { ids->usem.idsemt.vext->usem.offset = t1->usym.vsymt.saddr;
              ids->svtype = t1->usym.vsymt.vtypep;
              a = ids;
            }
          else a = ids->usem.bsemt.blexp;
         if (mexp)
            { bexps->usem.idsemt.vext->usem.offset = t2->usym.vsymt.saddr;
              bexps->svtype = t2->usym.vsymt.vtypep;
              b = bexps;
            }
          else b = bexps->usem.bsemt.blexp;
         gen_assign(a,b);
```

```
        if (mtarg) { t1 = t1->usym.vsymt.nxtvar;
                     if (t1 == NULL) break;  }
           else { ids = ids->usem.bsemt.blnext;
                  if (ids == NULL) break;  }
        if (mexp) { t2 = t2->usym.vsymt.nxtvar;
                    if (t2 == NULL) break;  }
           else { bexps = bexps->usem.bsemt.blnext;
                  if (bexps == NULL) break; }
    }  /* while true loop */
 }  /* gen_sassign */

/* ---------------------------------------------------------------------- */

eval_stor(semv, inst)
 semrecp semv;
 short inst;
  /* This generates a STOR operation on the object in SEMV. The
     object must be a simple variable, either a scalar or array */

 { symtype rt;
   symtabp sptr;

   if (!gencode)
     if (semv != NULL)
        if (semv->semt == IDENT)
           { sptr = semv->usem.idsemt.symp;
             if (sptr->symt == VARI && sptr->usym.vsymt.canchange &&
                 (sptr->rlevel == GLOBLEV || sptr->rlevel == rlevel) )
                       gen_var(semv, inst);
               else error("invalid assignment target");
           }
          else error("invalid assignment target");
        else error("EVAL_STOR disaster");
 }  /* eval_stor */

/* ---------------------------------------------------------------------- */

gen_procnum(port)
 semrecp port;
  /* Evaluate the semrec PORT and place the process number and the label
     expressions on the stack */

 { semrecp sp;
   int offset;  /* process number */

   if (!gencode)
     { offset = port->usem.idsemt.symp->usym.psymt.procnum;
       sp = port->usem.idsemt.vext;
       while (sp != NULL)
          { eval(sp->usem.arsemt.ax, INT_TYPE, NULL);
            sp = sp->usem.arsemt.arextnext;
          }
       gen_3byte(LCH, offset, port->usem.idsemt.symp->usym.psymt.dim);
     } /* if */
  }  /* gen_procnum */
```

```
/* ---------------------------------------------------------------------- */

 gen_rw( isread, list, i, mess, dim)
  boolean isread;
  semrecp list;
  int i, mess[], *dim;
   /* This generates code for a single input or output statement */

  { if (!gencode)
      if ( isread)
        { /* generate code for read statement */
         if ( (list->semt != IDENT) && (list->semt != SIG_NODE))
             error("non-simple variable in a READ");
           else /* simple variable or signal */
           { if (list->semt == SIG_NODE)
                { mess[i] = SGN;
                  mess[i+1] = list->usem.sigsemt.sigvar ?
                     list->usem.sigsemt.sigvar->usym.asymt.udim : 0; }
              else { mess[i]=wrtype(list->svtype->symt,TRUE);
                     mess[i+1] = list->svtype->usym.asymt.size; }
             if (list->semt != SIG_NODE) eval_stor(list,LID);
           }
         } /* end isread */

       else if (list->semt == STRNG)
          { /* string to output */
            eval(list,ARRAY_TYPE,list->svtype);
            mess[i] = ARR;
            mess[i+1] = strlen( &strtab[list->usem.stx]);
          }

        else /* must be simple write statement */
         { if (list->semt != SIG_NODE)
                eval(list, list->svtype->symt, list->svtype);
           if (list->semt == SIG_NODE)
              { mess[i] = SGN;
                mess[i+1] = list->usem.sigsemt.sigvar ?
                    list->usem.sigsemt.sigvar->usym.asymt.udim : NOSTRUCT;
               }
             else { mess[i]=wrtype(list->svtype->symt,TRUE);
                    mess[i+1] = list->svtype->usym.asymt.size; }
         } /* end write */
      *dim += 2;
   }  /* gen_rw */

/* ---------------------------------------------------------------------- */

 readwrite( isread, list, mess, dim)
  boolean isread;
  int *dim;
  int mess[];
  semrecp list;
   /* Calls gen_rw for each member of a structured read/write */

   { symtabp tt;

    if (list == NULL) return;
    if (list->semt != BLIST_NODE)
```

```
      { tt = list->svtype->usym.asymt.subtype;
        fix_messvar(list);
        rwrecur1(list, tt,isread,*dim,mess,dim);
      }
     else rwrecur2(list,isread,*dim,mess,dim);
   }  /* readwrite */

/* ---------------------------------------------------------------------- */

 rwrecur1( list, tt, isread,i, mess, no)
  semrecp list;
  symtabp tt;
  boolean isread;
  int mess[], *no, i;
   /* Enables last address/expression in message to placed onto stack first.
      This ensures that input/output occurs in the correct order.
      This function is used with message targets. */

  { semrecp a;

      if (tt == NULL) return;
      if (i > MAXMESS-2)
           { error("Too many message components"); return; }
        else rwrecur1(list, tt->usym.vsymt.nxtvar,isread, i+2, mess, no);
      list->usem.idsemt.vext->usem.offset = tt->usym.vsymt.saddr;
      list->svtype = tt->usym.vsymt.vtypep;
      gen_rw(isread,list,i,mess, no);
   }  /* rwrecur1 */

/* ---------------------------------------------------------------------- */

 rwrecur2( list, isread,i, mess, no)
  semrecp list;
  boolean isread;
  int mess[], *no, i;
   /* Same as rwrecur1 except used with structured vars and expressions */

   { semrecp a;

      a = list->usem.bsemt.blexp;
      if (list == NULL) return;
       else if (i > MAXMESS-2) { error("Too many message components");
                                 return;  }
         else rwrecur2(list->usem.bsemt.blnext,isread, i+2, mess, no);
      gen_rw(isread,a,i,mess, no);
  }  /* rwrecur2 */

/* ---------------------------------------------------------------------- */

 gen_guard(exp, failadr)
  semrecp exp;
  int failadr;
   /* Generates code for the boolean condition in a guard stmt,
      including the jump on false */

   { if (exp->semt != BOOLS)
           { eval(exp, BOOL_TYPE, boolptr);
             /* generate jump on false to fail address */
```

```
            gen_2byte(FJP,failadr);  }
        else if (exp->usem.numval == 0)
              { gen_2byte(UJP,failadr);
                if (!gencode) gencode++;  /* only increase if zero */
                /* suppress coding -- can never execute the commandlist */
              }
   }  /* gen_guard */

/* ------------------------------------------------------------------ */
```

APPENDIX E

```
#include <a:cspio.h>

 /* Intermediate code definitions */

  /* Arithmetic and logical ops */
#define ADI 1
#define ADR 2
#define SBI 3
#define SBR 4
#define MLI 5
#define MLR 6
#define DVI 7
#define DVR 8
#define MOD 9
#define NOT 10
#define NGI 11
#define NGR 12
#define EQI 13
#define EQR 14
#define GTI 15
#define GTR 16
#define LSI 17
#define LSR 18
#define AND 19
#define ORR 20

  /* Conversion operators */
#define FLO 21
#define FIX 22

  /* Input and Output */
#define MSI 23
#define MSO 24

  /* Guard delimiters */
#define GDB 25
#define GDE 26
#define EDF 27    /* marks the end of otherwise guard */

  /* Jump operators */
#define ALN 28    /* alternate stmt */
#define AJP 29
#define AJD 30    /* jump table entry for otherwise guard */
#define FJP 31
#define UJP 32

  /* Store ops */
#define STI 33
#define STR 34
#define STA 35
#define SOI 36    /* NB. the three SOx codes must follow directly */
#define SOR 37    /* after the three STx codes */
#define SOA 38

#define STLD 3
```

```
  /* Load ops */
#define LDI 39
#define LDR 40
#define LDA 41
#define LOI 42
#define LOR 43
#define LOA 44
#define LDS 45   /* string */
#define LSG 46   /* signal */
#define LCH 47

#define LID 48   /* load address */
#define IND 49   /* check index */
#define PSI 50   /* push integer */
#define PSR 51

  /* Miscellaneous */
#define LOC 52
#define DEF 53
#define LAB 54   /* load process labels */
#define INC 55
#define DEC 56
#define BEG 57
#define EXT 58   /* end of a process */
#define HLT 59
#define JBL 60   /* start of jump table */
#define EBL 61   /* end of jump table */
#define CRG 62
#define NOP 63

  /* Debugging codes -- not emitted by compiler */
#define STK 64   /* dump execution stack */
#define DMP 65   /* dump process ring */

#define LASTCODE 65

  /* Possible states for program execution -- value of PS */
#define FINISHED 0
#define RUNNING 1
#define DIVCHK 2
#define EOFCHK 3
#define IOCHK 4
#define STKCHK 5
#define RANCHK 6
#define DEDCHK 7
#define CODCHK 8

  /* States of process execution */
#define READY 0
#define SUSP 1
#define TERMINATED 2
#define QUERY 3
#define IDLE 4
#define WAIT 5        /* wait if communicant is in query state */
#define USELESS 6     /* when only otherwise guard can be executed */
```

```
  /* State of guard once tested --
     that is value of FAILED is true, false or pending */
#define PENDING -1

  /* Variable Limits */
#define STACKMAX 10000    /* execution stack */
#define MAXPART 150       /* maximum stack allocated to each process */
#define CODEMAX 1000      /* intermediate code array */
#define FILELEN 12        /* maximum length of filenames */
#define LEVELS 5          /* maximum depth of process nesting */
#define MAXLAB 5          /* maximum dimension of process labels */
#define LABELMAX 100      /* maximum number of labels used in code file */
#define MAXMESS 20        /* no * 2 components for message passing */

  /* Other constants */
#define WRITE_MODE  "w"
#define READ_MODE  "r"
#define LINELEN 80         /* maximum length of a line */
#define EOS '\0'           /* marks end of a string */
#define EOLN '\n'
#define REALSIZE 4         /* number of 2-byte stack spaces used for doubles */
#define GLOBAL 1           /* outer process runtime level */
#define IOSTD 1            /* larger standard i/o process number */
#define INP 0              /* input synchronization */
#define OUP 1              /* output synchronization */

  /* Used for I/O functions */
#define ITG 0
#define REL 1
#define ARR 2
#define SGN 3
#define CHR 4
#define MSS 5


  /* Structure for guard nodes -- eventually linked into a ring */
typedef struct grdnode
      { int jumplab;  /* new pc value -- address of corresponding guard */
        struct grdnode *gnext;
        boolean failed; /* status of input guards */
        union { struct
                { int nodetot;   /* number of nodes in ring excl. dummy */
                  int tested,    /* no. of guards tested in this round */
                      failtest;  /* no. of guards failed */
                  int failadr;   /* to identify each guard ring */
                  struct grdnode
                         *prevhead, /* previous guardring ptr */
                         *prevcur;  /* previous curguard ptr */
                 }  dumnode;
              struct  { int glabs[MAXLAB];  /* copy of guard labels */
                        int dm;      /* dimension of labels */
                      } altnode;
            }  grdun;
       } *grdptr;
```

```
  /* Other dynamic structures */
typedef struct asknode
          { struct asknode *more;       /* next process */
            struct descript *askid;     /* querying this process */
          } *aptr;

typedef struct qrynode
          { struct qrynode *more;       /* next process */
            struct descript *queryid;   /* queried by this process */
            grdptr querypc;             /* guard at which to resume execution */
            int qformat[MAXMESS];       /* format of query */
            int querydirect;            /* input or output specified */
           } *qptr;

typedef struct descript
      {  struct descript *next,          /* next process in ring */
                         *parent;        /* outer process */
         struct
             { struct descript *mate;    /* pointer to other half */
               short direc;              /* either I or O for this process */
               int format[MAXMESS];      /* format for message passing */
             } inout;                    /* linked waiting on I/O */
         short int status;       /* process status flag */
         int dim;                /* no of label dimensions */
         int lev;                /* runtime level */
         int num, unid;          /* process number and unique id */
         int sibs;               /* number of subordinate processes */
         int labs[MAXLAB];       /* label identification */
         int guardlbl;           /* used for inputs in guards -- failadr */
         grdptr guardring,       /* dummy node in guard ring */
                curguard,        /* current guard being tested */
                inrupt;          /* next guard after executing otherwise */
         aptr  askhead;          /* list of processes queried by me */
         qptr  queryhead;        /* list of processes querying me */
         int firstinst,          /* first code address */
             lastio;             /* address of last MSO/MSI instruction */
         int ppc, ppb, ppt;      /* prog count, base, stk ptr */
         int stackend;           /* memory limit */
         int display[LEVELS];    /* needed for runtime levels */
      } *procptr;

typedef union flint
          { int intval[REALSIZE];
            double flval;
          } realint;

typedef struct lptr
          { int padr;  /* code byte to patch */
            struct lptr *nxtpatch;
          } *labptr;

typedef struct lbls
          { int labval;
            boolean defined;
            labptr patch;
          } table;
```

```
/* Declarations for the CSP-i interpreter */

#include "CSPIDEFS.C"

 FILE *dfile, *tfile, *rfile;  /* data, trace and results file */
 boolean inpstd = FALSE;       /* data file != std input yet */

/* Intermediate code array and pointers */
 int code[CODEMAX];           /* to store the ILC's - automatic initialisation */
 int nxtcode = 0;             /* pointer to next free code space */

/* Execution stack variables */
 int stk[STACKMAX];           /* automatically initialised to zero */
 int tos = -1;                /* marks top of stack */
 int partition = 75;          /* gives stack partition for each process */

/* Intermediate backpatch table */
 table labtbl[LABELMAX];

/* Interpreter variables */
 int pc = -1;            /* Program counter */
 short ps = RUNNING;     /* Program status */
 procptr  firstproc = NULL,  /* pointer to first process descriptor */
          current = NULL,    /* process currently executing */
          globptr = NULL,    /* program process -- num = -1 */
          istd = NULL;       /* standard input process -- num = 0 */

 boolean tracing = FALSE; /* whether a trace must be created */
 int totprocs = 1;      /* total number of concurrent procs defined -- allows
                           for input and output procs as well-- but these do
                           not feature in the allocation of stack space */
 int readyprocs = 0;    /* number of procs ready to go */
 int seed = 11;         /* random number generator seed */

/* Debugging variables */
 char *ilc[] = { "junk", "ADI", "ADR", "SBI", "SBR", "MLI", "MLR", "DVI",
                 "DVR", "MOD", "NOT", "NGI", "NGR", "EQI", "EQR", "GTI",
                 "GTR", "LSI", "LSR", "AND", "ORR", "FLO", "FIX", "MSI",
                 "MSO", "GDB", "GDE", "EDF", "ALN", "AJP", "AJD", "FJP",
                 "UJP", "STI", "STR", "STA", "SOI", "SOR", "SOA", "LDI",
                 "LDR", "LDA", "LOI", "LOR", "LOA", "LDS", "LSG", "LCH",
                 "LID", "IND", "PSI", "PSR", "LOC", "DEF", "LAB", "INC",
                 "DEC", "BEG", "EXT", "HLT", "JBL", "EBL", "CRG", "NOP",
                 "STK", "DMP" };

/* Used in process dumps to reflect the state of a process */
  char statstate[] = {'R', 'S', 'T', 'Q', 'I', 'W', 'U'};
```

```
/* External variable declarations and function definitions */

/* General Utilities */
  extern int strlen();        /* returns length of a string */
  extern int strcmp();        /* is s1 == s2 ? (0, +, -)  */
  extern char *strcpy();      /* copies s2 to s1  */
  extern char *strcat();      /* catenates s2 to end of s1 */
  extern FILE *fopen();       /* returns file pointer */
  extern int fclose();        /* closes a file */
  extern fprintf(), fputc(), printf();       /* output routines */
  extern int fscanf(), sscanf();
  extern char *gets(), *fgets();             /* input routines  */
  extern int rlsmem();        /* release memory */
  extern long sizmem();       /* gives size of free memory pool */

/* Interpreter variables */
  extern int pc;                 /* Program counter */
  extern short ps;               /* Program status */
  extern procptr firstproc,      /* pointer to first process descriptor */
                 current,        /* process currently executing */
                 globptr,        /* program process -- num = -1 */
                 istd;           /* standard input process -- num = 0 */
  extern boolean tracing;     /* whether a trace must be created */
  extern int totprocs;        /* total number of concurrent procs defined */
  extern int readyprocs;      /* number of procs ready to go */
  extern int seed;            /* random number generator seed */
  extern FILE *dfile, *tfile, *rfile;  /* data, trace and results files */

/* Intermediate code array and pointers */
  extern int code[];          /* to store the ILC's */
  extern int nxtcode;         /* pointer to next free code space */

/* Execution stack variables */
  extern int stk[];
  extern int tos;             /* marks top of stack */
  extern int partition;

/* Intermediate label array and backpatch table */
  extern table labtbl[];

/* Used in process dumps to reflect the state of a process */
  extern char statstate[];

/* Debugging variables */
  extern char *ilc[];
```

```
/* CSP interpreter -- general routines for reporting errors, allocating
   memory, creating the label table and more */

#include "CSPIDEFS.C"
#include "CSPIEXT.C"

extern char *getmem();

/* ---------------------------------------------------------------------- */

 procptr newproc()
   /* Allocate memory for a process descriptor */
   { char *new;
     if ( !(new = getmem(sizeof (struct descript))))
            abort("Out of memory!");
      else return((procptr) new);
    }  /* newproc */

/* ---------------------------------------------------------------------- */

 labptr newpatch()
   /* Allocate memory for another label backpatch entry */
   { char *new;
     if ( !(new = getmem (sizeof (struct lptr)))) abort("Out of memory!");
      else return((labptr) new);
   }  /* newpatch */

/* ---------------------------------------------------------------------- */

 grdptr getguard()
   /* Allocate memory for a guard entry */
   { char *new;
     if ( !(new = getmem (sizeof (struct grdnode)))) abort("Out of memory!");
          else return((grdptr) new);
   }  /* getguard */

/* ---------------------------------------------------------------------- */

 aptr newanode()
   /* Allocate memory for an asknode */
   { char *new;
     if ( !(new = getmem (sizeof (struct asknode)))) abort("Out of memory!");
       else return((aptr) new);
    }  /* newanode */

/* ---------------------------------------------------------------------- */

 qptr newqrynode()
   /* Allocate memory for a querynode */
   { char *new;
     if ( !(new = getmem (sizeof (struct qrynode)))) abort("Out of memory!");
        else return((qptr) new);
    }  /* newqrynode */

/* ---------------------------------------------------------------------- */

 static char resp(msg)
   char msg[];
```

```
    /* Print a message and return a single character response. */

   { char ch, chstring[80];

    fprintf(stderr, "%s", msg);
    ch = (gets(chstring)) ? chstring[0] : '\n';
    putc('\n', stderr);
    return(ch);
   }  /* resp */

/* ---------------------------------------------------------------------- */

 boolean yesresp (msg)
  char msg[];
   /* Query with a Y or N reply */

   { char ch;

    ch = resp(msg);
    return((ch == 'y') || (ch == 'Y'));
  }  /* yesresp */

/* ---------------------------------------------------------------------- */

 static more(msg)
  char msg[];
   /* Print the string, and let the user type any character to proceed */
  { char foo = resp(msg); }  /* more */

/* ---------------------------------------------------------------------- */

 abort(msg)
  char msg[];
  { fprintf(stderr, "ABORT! %s\n" ,msg);
    fprintf(stderr, "Process = %d pc = %d base = %d top = %d\n",
            current->num, current->ppc, current->ppb, current->ppt);
    if (tracing) dumpprocs();
    exit(1);
  }  /* abort */

/* ---------------------------------------------------------------------- */

 error(msg)
  char msg[];
  { fprintf(stderr, "ERROR: %s\n",msg);
    more("Type any character to continue: ");
  }  /* error */

/* ---------------------------------------------------------------------- */

 insertdef(labno, valu)
  int labno, valu;
   /* Define a label -- inserting it into the labtbl */

   { labptr nxt,remv;

     labno %= LABELMAX;
     labtbl[labno].defined = TRUE;
```

```
    labtbl[labno].labval = valu;
    nxt = labtbl[labno].patch;
    labtbl[labno].patch = NULL;
    while (nxt != NULL)  /* fill in forward references */
     { code[nxt->padr] = valu;
       remv = nxt;
       nxt = nxt->nxtpatch;
       if (rlsmem ( (char *) remv, sizeof (struct lptr)) < 0)
        error ("Release memory incomplete");
     }
  }  /* insertdef */

/* ---------------------------------------------------------------------- */

static int seed = 11;
#define ITWO 2

 int urand(ival)
  int ival;
   /* Based on Knuth's random number generator */

  { static int ia, ic, mic;
    static int m2;
    float temp;
    int m;
    double halfm, atan(), sqrt();
    static float s;

    if (m2 == 0)
     /* if first entry, compute machine integer word length */
     { m = 1;
       do { m2 = m; m = ITWO * m2; }
         while (m > m2);
       halfm = m2;
       /* compute multiplier and increment for linear congruential method */
       ia = 8* (int) (halfm*atan(1.0) / 8.0) + 5;
       ic = 2* (int) (halfm*(0.5 - sqrt(3.0) / 6.0)) + 1;
       mic = (m2 - ic) + m2;
         /* s is the scale factor for converting to floating point */
       s = 0.5 / halfm;
     }
     /* compute next random number */
   seed *= ia;
     /* for computers that do not allow integer overflow on addition */
   if (seed > mic) seed = (seed - m2) - m2;
   seed += ic;
     /* following statement for computers where the word length for
        addition is greater than for multiplication */
   if (seed/2 > m2) seed = (seed - m2) - m2;
     /* following statement for computers where integer overflow affects
        the sign bit */
   if (seed < 0) seed = (seed + m2) + m2;
   temp = seed * s * ival;   /* gives float in the range 0 and ival */
   return( (int) temp);
  }  /* urand */

/* ---------------------------------------------------------------------- */
```

```
/* Debugging routines for interpreter */

#include "CSPIDEFS.C"
#include "CSPIEXT.C"

/* --------------------------------------------------------------------------- */

 dumpstack(x)
  procptr x;
   /* Dump the contents of the stack for the process (x); also dump
      contents of the display */

   { int i;

     if (!tracing) return;
     fprintf(tfile, "Display dump: ");
     for ( i=0; i<=x->lev; i++) fprintf(tfile, "%5d ", x->display[i]);
     fprintf(tfile, "\nStack dump at %d process = %d; top = %d base = %d \n",
             current->ppc-1, current->num, current->ppt, current->ppb);
     for ( i=current->ppb; i<=current->ppt; i++)
       fprintf(tfile, "%5d%c", stk[i], (i % 8) ? ' ' : '\n');
     putc('\n',tfile);
   }  /* dumpstack */

/* --------------------------------------------------------------------------- */

  dumpprocs()
   /* Prints the descriptors of all processes in the ring */

   { procptr x = firstproc;
     int i;

     if (!tracing) return;
     fprintf(tfile, "Id  Num Lev Dim   Labels  Rdy  Par Inout Sibs ppc ");
     fprintf(tfile, "ppb ppt stkend  display\n");

     if (x)
     do
      { fprintf(tfile, "%3d %3d %3d %2d   ", x->unid, x->num, x->lev, x->dim);
        for (i=0;i<MAXLAB; i++) fprintf(tfile, "%ld ", x->labs[i]);
        fprintf(tfile, "%2c %4d %4d %4d %3d %3d %3d %5d",
                statstate[x->status], (x->parent) ? x->parent->unid : 99,
                (x->inout.mate) ? x->inout.mate->unid : 99, x->sibs,
                x->ppc, x->ppb, x->ppt, x->stackend);
        for (i=0;i<LEVELS; i++) fprintf(tfile, " %2d", x->display[i]);
        putc('\n', tfile);
        x = x->next;
       } while (x != firstproc && x != NULL);
   }  /* dumpprocs */

/* --------------------------------------------------------------------------- */
```

```
 /* CSP interpreter -- main function and initialization */

#include "CSPIDEFS.C"
#include "CSPIEXT.C"

extern procptr search();
extern procptr newproc();
static FILE *cfile;
extern boolean yesresp(), inpstd;

/* Lexical variables */
 static int ch;              /* next character from input file */

/* ---------------------------------------------------------------------- */

 static openfiles()
  /* Opens code, data, trace and results files */

   { int fillen;
     boolean success;
     char filename[FILELEN];

    /* first, get the code file */
    do {
      printf("What code file? ");
      if (gets(filename) && (strcmp(filename, "") != 0))
        { fillen = strlen(filename);
          if ( (fillen < 4) || (filename[fillen-4] != '.'))
                 strcat(filename,".cod");
          cfile = fopen(filename, READ_MODE);
        }
      success = (cfile != NULL);
      if (!success)
        printf("file doesn't exist; try again\n");
    } while (!success);

   /* Get data file; defaults to the console */
    do {
      printf("What data file? ");
      if (gets(filename) && (strcmp(filename, "") != 0))
        { fillen = strlen(filename);
          if ( (fillen < 4) || (filename[fillen-4] != '.'))
                 strcat(filename,".dat");
          dfile = fopen(filename, READ_MODE);
        }
       else { dfile = fopen("con:", READ_MODE);
              inpstd = TRUE; }

      success = (dfile != NULL);
      if (!success)
        printf("file doesn't exist; try again\n");
    } while (!success);

    /* now, get the trace file; defaults to console */
    if (tracing || (tracing = yesresp("Trace? ")) )
     { do
        { printf("What trace file? ");
          if (gets(filename) && (strcmp(filename, "") != 0))
```

```
          { tfile = fopen(filename, READ_MODE);
            success = (tfile == NULL);
            fclose(tfile);
            if (!success)
               success = yesresp("..already exists, purge it? ");
          }
        else
         { strcpy(filename, "con:");
           success = -1;  }
       } while (!success);
    tfile = fopen(filename, WRITE_MODE);
   } /* if */

   /* get results file; defaults to console: */
   do {
     printf("What results file? ");
     if (gets(filename) && (strcmp(filename, "") != 0))
       { fillen = strlen(filename);
         if ( (fillen < 4) || (filename[fillen-4] != '.'))
              strcat(filename,".txt");
         rfile = fopen(filename, READ_MODE);
         success = (rfile == NULL);
         fclose(rfile);
         if (!success)
            success = yesresp("..already exists, purge it? ");
       }
     else { strcpy(filename, "con:");
            success = -1; }
   } while (!success);
    rfile = fopen(filename, WRITE_MODE);
  }  /* openfiles */

/* ---------------------------------------------------------------------- */

 static postmortem()
  /* Report any error condition found once execution halts */

   { fprintf(rfile, "\n ****** \n");
     switch (ps)
      { case DIVCHK: fprintf(rfile,"Division by zero ");
                     break;
        case EOFCHK: fprintf(rfile,"No more data ");
                     break;
        case IOCHK:  fprintf(rfile,"Input / Output error ");
                     break;
        case STKCHK: fprintf(rfile,"Stack overflow ");
                     break;
        case RANCHK: fprintf(rfile,"Range error ");
                     break;
        case CODCHK: fprintf(rfile,"Invalid code file ");
                     break;
        case DEDCHK: fprintf(rfile,"Deadlock! ");
                     break;
      }
     fprintf(rfile, " in process %d \n", current ? current->num : -1);
     dumpprocs();
   }  /* postmortem */
```

```
/* ---------------------------------------------------------------------- */

 static int getinst()
  /* Gets the next inst from code file; ignores comment lines */

   { int val, noread;

      if (nxtcode >= CODEMAX) abort("Codefile overflow! ");
      if ( (noread = fscanf(cfile,"%d%c", &val, &ch)) != EOF && noread == 2)
               return(code[nxtcode++] = val);
        else if (noread != EOF)
              { do ch = getc(cfile); while (ch != '\n' && ch != EOF);
                return(getinst()); }
        else return(0);
    }  /* getinst */

/* ---------------------------------------------------------------------- */

 static int nextval(byteval)
  boolean byteval;
   /* Gets next value from line and inserts into code array */

  { int i;
    float vall;
    double *dptr;

    if (nxtcode >= CODEMAX) abort("Codefile overflow! ");
    if (byteval) fscanf(cfile, "%d%c", &code[nxtcode++], &ch);
      else { fscanf(cfile,"%f%c", &vall, &ch);
             dptr = (double *) &code[nxtcode];
             *dptr = vall;
             nxtcode += REALSIZE;
           }
    return(code[nxtcode-1]);
  }  /* nextval */

/* ---------------------------------------------------------------------- */

 int getnum()
  /* Gets next integer from code line, but does not insert
     this value into the code array */

  { int ival;

    fscanf(cfile, "%d%c", &ival, &ch);
    return(ival);
  }  /* getnum */

/* ---------------------------------------------------------------------- */

 static readcode()
  /* Reads in intermediate code from designated code file and sets up
     the process descriptors for outer level of processes. */

  { int previnst = 0,    /* previous instruction */
        inst;            /* current instruction */
    int i;
```

```
    while (inst = getinst())
      { switch (previnst = inst)
          { case PSR: i = nextval(FALSE); break;
            case BEG: createproc(); break;
            case HLT: globptr->ppc = nxtcode-1;
                      i = getnum(); /* read last byte */ break;
            case LOC: insertdef(getnum(), --nxtcode); break;
            case DEF: nxtcode--;    /*  not a code to be executed */
                      i = getnum();
                      insertdef(i, getnum()); break;
            case LAB: abort("LAB - unexpected instruction"); break;
            case FJP:  /* these 2 byte instructions use labels */
            case JBL:
            case UJP:
            case EDF:
            case GDB:
            case ALN:
            case AJD:
            case INC: code[nxtcode] = lookup(getnum(),nxtcode++);
                      break;
            case AJP: code[nxtcode] = lookup(getnum(),nxtcode++);
                      /* now do default as well */
            default:  while (ch != EOLN) i = nextval(TRUE); break;
          }
      }
      /* allocate memory partition for each process */
      /* actually total num of procs + 1 */
    if ( (partition = STACKMAX / totprocs) > MAXPART) partition = MAXPART;

   /* for debugging purposes */
    if (tracing)
     { for (i=0;i<nxtcode;i++)
        { if (! (i%10) ) fprintf(tfile, "\n %-5d:: ", i);
          fprintf(tfile,"%-5d  ", code[i]);
        }
       fputc('\n', tfile);
     }
    if (previnst != HLT) abort("Incomplete code file");
  }  /* readcode */

/* ---------------------------------------------------------------------- */

 static initouter()
  /* Allocate stack space for outer processes,  and initialise static
     link and return address. Also set the appropriate descriptor fields.
     Finalises the ring by linking the last process to the first. */

   { procptr x = firstproc,
             last = firstproc;
     boolean foundglobal = FALSE;

      while (x != NULL)
       { if (x->lev == GLOBAL) { allocate(x); globptr->sibs++;
                                 foundglobal = TRUE; }
         last = x;
         x = x->next;
       }
      if (foundglobal) { readyprocs--; globptr->status = SUSP; }
```

```
      last->next = firstproc;
      current = last;
    }  /* initouter */

/* ---------------------------------------------------------------------- */

 procptr findready(start)
  procptr start;
   /* Searches for the next ready process in the process ring, starting at
      a given point */

   { procptr x = start;

      if (start == NULL) return(NULL);
      if (!x->status || x->status == QUERY || x->status == USELESS)
         return(x);
      while (x->next != start)
       { if (!x->next->status || x->next->status == QUERY
              || x->next->status == USELESS)  return(x->next);
         x = x->next;
       }
      return(NULL);  /* no ready procs found */
   }  /* findready */

/* ---------------------------------------------------------------------- */

#define SMALLNUM 10

 static interpret()
  /* Runs the interpreter */

   { int rnum, urand(), dumpstack(), dumpprocs();
     char c;

     initouter();
     do
       { rnum = urand(SMALLNUM);
         if  ((current = findready(current->next)) == NULL) ps = DEDCHK;
            else
               if (current == istd && readyprocs == 1 && !current->queryhead)
                    ps = DEDCHK;
         if (tracing) fprintf(tfile, "New process = %d, labl = %d \n",
                             current->num, current->labs[0]);
         while (ps == RUNNING && rnum-- > 0 && (!current->status ||
                current->status == QUERY || current->status == USELESS) )
           {  if (current->ppc < 0) { fudgeinput(); rnum = 0; }
                   else nextstep();
           }
       }
     while (ps == RUNNING);
     if (ps != FINISHED) postmortem();
    }  /* interpret */

/* ---------------------------------------------------------------------- */

 static initstdi()
  /* Initialise the process descriptor for the standard input process */
```

```
 { istd = newproc();
   onedesc(istd);
   istd->next = firstproc;
   firstproc = istd;
   istd->num = istd->unid = INP;
   istd->status = READY; readyprocs++;
   istd->firstinst = istd->ppc = -1;
 }  /* initstdi */

/* ---------------------------------------------------------------------- */

static initinterp()
 /* Set up globptr process descriptor and do initialization */

 { int i;

    for ( i =0; i<LABELMAX; i++) labtbl[i].defined = FALSE;
    globptr = newproc();
    onedesc(globptr);   /* initialise fields */
    globptr->num = globptr->unid = -1;
    firstproc = globptr;
    globptr->next = NULL;
    globptr->status = READY;  readyprocs++;
    globptr->lastio = -1;
    globptr->firstinst = 0;
    initstdi();
  }  /* initinterp */

/* ---------------------------------------------------------------------- */

main(argc, argv)
 int argc;
 char *argv[];

{ if (argc > 1) tracing = TRUE;  /* set tracing level */
  printf("CSPINT [an interpreter for CSP-i]  vs. 1-Nov-86]\n");
  openfiles();
  initinterp();
  readcode();
  interpret();
  fclose(cfile);
  if (tracing) { fflush(tfile);  fclose(tfile); }
  fflush(rfile);
  fclose(rfile);
}  /* main */

/* ---------------------------------------------------------------------- */
```

```
/* CSP-i interpreter -- general procedures */

#include "CSPIDEFS.C"
#include "CSPIEXT.C"

extern int rlsmem(), getnum();
extern procptr newproc();
extern labptr newpatch();
extern procptr partner();

/* ---------------------------------------------------------------------- */

onedesc(ptr)
 procptr ptr;
  /* Sets up a frame for a process descriptor */

  { int i;

    ptr->sibs = 0;    /* no subordinate processes active */
    ptr->dim = ptr->lev = ptr->ppc = 0;
    ptr->ppb = ptr->ppt = ptr->stackend = 0;
    ptr->inrupt = NULL;
    ptr->inout.mate = ptr->parent = NULL;
    ptr->guardlbl = ptr->inout.direc = -1; /* no guards; no inout */
    for (i=0; i<LEVELS; i++) ptr->display[i] = 0;
    for (i=0; i<MAXLAB; i++) ptr->labs[i] = 0;
    ptr->guardring = ptr->curguard = NULL;
    ptr->askhead = NULL; ptr->queryhead = NULL;
    ptr->status = SUSP; ptr->lastio = -1;
   }  /* onedesc */

/* ---------------------------------------------------------------------- */

static copy_rec(dst, src)
 procptr src, dst;
  /* Copies one process descriptor to another */

 { char *s = (char *) src, *t = (char *) dst;
   int size = sizeof(struct descript);
   int i = 0;

   while (i++ <= size) *t++ = *s++;
 }  /* copy_rec */

/* ---------------------------------------------------------------------- */

procptr search(no, from)
 int no;
 procptr from;
  /* Return a pointer to the first occurrence, after the process (from),
     of the process with given number (no); uses a ring of process
     descriptors */

  { procptr x = from->next;

      if (from->num == no) return(from);
      while (x != firstproc)
       { if (x->num == no) return(x);
```

```
        x = x->next;
      }
    return(NULL);  /* no process descriptor found */
  }  /* search */

/* ---------------------------------------------------------------------- */

procptr match(num)
 int num;
  /* Find first process with procnum matching arguments.  Uses a linked
     list; NOT a ring, so that is can only be used before initouter()
     is called. */

 { procptr found = firstproc;

    while (found != NULL)
      { if (found->num == num) return(found);
        found = found->next;
      }
    return(NULL);
  }  /* match */

/* ---------------------------------------------------------------------- */

allocate (ptr)
 procptr ptr;
  /* Allocate heap space for a process (ptr) and initialise appropriate
     fields of descriptor and stack */

  { int i;

     ptr->ppc = ptr->firstinst;    /* reset program counter */
     ptr->ppb = tos+1;    /* new base for process */
     ptr->ppt = tos;      /* stack space used by process */
     tos += partition;
     if ((ptr->stackend = tos) > STACKMAX)
       abort("Too little stack space"); /* end of stack area for process */
     ptr->display[ptr->lev] = ptr->ppb;  /* insert new display level */
     stk[ptr->ppb] = ptr->display[ptr->lev-1];  /* static link */
     stk[ptr->ppb+1] = 0;    /* return address */
     ptr->status = READY; readyprocs++;
       /* store values for the label variables */
     for (i=0; i<ptr->dim; i++) stk[ptr->ppb+2+i] = ptr->labs[i];
  }  /* allocate */

/* ---------------------------------------------------------------------- */

reinit (ptr)
 procptr ptr;
  /* Reinitialize the fields of a process descriptor if the process is
     activated for a second or subsequent time */

  {  ptr->ppc = ptr->firstinst;    /* reset program counter */
     ptr->ppt = ptr->ppb-1;        /* stack space used by process */
     ptr->status = READY; readyprocs++;
  }  /* reinit */
```

```
/* ----------------------------------------------------------------------- */

static struct alabl { int up, low; };

static creatcopy(labarr, prosess, ancest)
 int labarr[ ];
 procptr ancest, prosess;
  /* Set up the new descriptor with labels given by (labarr) */

  { int i;
    procptr current;

    current = newproc( );
    copy_rec(current, prosess);
    current->parent = ancest;
    for (i=0; i<MAXLAB; i++)  current->labs[i] = labarr[i];
    prosess->next = current;
    current->unid = ++totprocs;
  }  /* creatcopy */

/* ----------------------------------------------------------------------- */

static multiproc(labbs, onepr, dims, done, copy, ancest)
 struct alabl labbs[ ];
 int onepr[ ], done, dims;
 procptr ancest, copy;
  /* Create copies of the process descriptor (COPY), filling in
     unique labels. This function merely calculates the relevant
     labels, and then calls creatcopy( ). */

  { int i;

    if (done == dims)  { creatcopy(onepr,copy,ancest); return; }
    for (i=labbs[done].low; i<=labbs[done].up; i++)
        { onepr[done] = i;
          multiproc(labbs, onepr, dims, done+1, copy, ancest); }
  }  /* multiproc */

/* ----------------------------------------------------------------------- */

createproc( )
 /* Setup another process descriptor and link into ring; new processes
    are added in at the head of the chain */

 { procptr x,temp;
   int parno;      /* number of parent process */
   int i;
   int vals[MAXLAB];
   struct alabl labels[MAXLAB];

  temp = newproc( );
  onedesc(temp);    /* initialise attributes */
  temp->next = firstproc;
  firstproc = temp;
  temp->num = getnum( );   /* read process number */
  temp->lev = getnum( );   /* runtime level */
  parno = getnum( );      /* num of parent process */
  temp->parent = (parno > 0) ? match(parno) : globptr;
```

```
  temp->firstinst = nxtcode;  /* first instruction for this process */
  for (i=0;i<MAXLAB;vals[i++]=0) ;
  temp->dim = getnum();   /* number of process labels */
  for (i=0; i<temp->dim; i++)      /* labelled process */
      { if  (getnum() != LAB) abort("Error in code file - LAB expected");
            else { labels[i].low = getnum();  /* lower limit */
                   labels[i].up = getnum();   /* upper limit */
                 }
      }
     /* make new proc descriptors for the number of instances */
  x = temp->parent;
  if (temp->dim)  /* ie. only labelled processes */
     {  do
           { multiproc(labels,vals,temp->dim,0,temp,x);
             if (x != NULL) x = x->next;
           } while (x != temp->parent && x->num == parno);
        firstproc = temp->next;
        if (rlsmem ((char *) temp, sizeof (struct descript)) < 0)
            /* remove skeleton process */
              error ("Memory release unsuccessful");
      }
    else temp->unid = ++totprocs;
 }  /* createproc */

/* ---------------------------------------------------------------------- */

int lookup(lab, patchadr)
 int lab, patchadr;
  /* Substitute the value of a label if defined, else add a new
     address to the label's backpatch list */

  { labptr temp;

      if (lab < 0) return(lab);
      lab %= LABELMAX;
      if (labtbl[lab].defined) return(labtbl[lab].labval);
       else { temp = newpatch();
              temp->nxtpatch = labtbl[lab].patch;
              labtbl[lab].patch = temp;
              temp->padr = patchadr;
              return(0);
            }
  }  /* lookup */

/* ---------------------------------------------------------------------- */

callproc (old)
 procptr old;   /* calling routine */
  /* Start processes with parent (old) -- allocate space and suspend
     parent process */

   { procptr x = firstproc;
     int i;
     boolean found = FALSE;

     do { if (x->parent == old)
          { if (x->status == SUSP)
              /* initialising for the first time */
```

```
            { for (i=0; i<x->lev; i++) x->display[i] = old->display[i];
              allocate(x); }
             else reinit(x);
            found = TRUE;
            old->sibs++;  /* add one more subordinate process */
          }
        x = x->next;
      } while (x != firstproc);
    if (!found) error("Disaster in call - no descriptor found");
    old->status = SUSP; readyprocs--;
  }  /* callproc */

/* --------------------------------------------------------------------- */

removeproc(ptr)
 procptr ptr;
  /* "Remove" a terminated process from descriptor ring -- does not
      actually unlink descriptor, because it can be reactivated */

  { procptr x = firstproc;
    qptr oldhead;
    aptr ahead, prev;

    readyprocs--;
    if (ptr->parent) /* reset number of children of parent */
      { if (!--ptr->parent->sibs)
            {readyprocs++; ptr->parent->status = READY; }
        if (ptr->parent->ppc < ptr->ppc) ptr->parent->ppc = ptr->ppc;
      }
    ptr->status = TERMINATED;
    do  /* find any other processes suspended on this one */
     { if (x->status == SUSP && x->inout.mate == ptr)
          { x->status = READY; x->inout.mate = NULL;
            readyprocs++; collapse(x, x->inout.direc, x->inout.format);
          }
      x = x->next;
     } while (x != firstproc);
    while (oldhead = ptr->queryhead)
     { oldhead->querypc->failed = TRUE;
       oldhead->queryid->guardring->grdun.dumnode.failtest++;
         /* Now remove ask id from non-terminating process */
       ahead = oldhead->queryid->askhead; prev = NULL;
       while (ahead && ahead->askid != ptr)
          { prev = ahead; ahead = ahead->more; }
       if (ahead)
           { if (prev) prev->more = ahead->more;
                  else oldhead->queryid->askhead = ahead->more;
             if (rlsmem( (char *) ahead, sizeof (struct asknode)) < 0)
                  error ("Incomplete memory release - asknode");
           }
         /* put idle processes in query mode */
       if (oldhead->queryid->status == IDLE)
         { readyprocs++; oldhead->queryid->status = QUERY; }
       ptr->queryhead = ptr->queryhead->more;
       if (rlsmem( (char *) oldhead, sizeof (struct qrynode)) < 0)
           error("Incomplete memory release -- qrynode");
     }
  }  /* removeproc */
```

```
/* ------------------------------------------------------------------------ */

 dectby( i)
  int i;
   /* Decrement the stack pointer */

   { if ( (current->ppt -= i) < current->ppb) abort ("Stack underflow"); }

/* ------------------------------------------------------------------------ */

 incby( i)
  int i;
   /* Increment the stack pointer of the current process */

   { if ((current->ppt += i) > current->stackend) abort("Stack overflow");  }

/* ------------------------------------------------------------------------ */

 int nextcode ()
   /* Fetch the next code/data from code array incrementing the ppc
      of the current process */

   { return( code[current->ppc++]); }  /* nextcode */

/* ------------------------------------------------------------------------ */

 nextstep( )
  /* Interprets the individual intermediate codes */

   { register i;
     int dm,            /* temporary variables */
         inst,          /* gives current instruction */
         l, off,        /* relative level, and offset */
         size,
         urand( ),      /* random number generator */
         chk;           /* catches any errors in I/O */
     procptr ioproc;    /* keeps track of process number for I/O */
     double *dptr1, *dptr2;

    inst = nextcode( );
    if (tracing) fprintf(tfile, "Opcode = %s\n", ilc[inst]);
    switch (inst)
     { case GDB:
          current->guardlbl = nextcode( ); break;
       case GDE:
          if (current->status != USELESS)  /* default guard stays useless */
             current->status = READY;      /* no longer in query mode */
          sigwaits(current);
          current->guardlbl = -1;
          break;
       case ADI:
          dectby( 1); stk[current->ppt] += stk[current->ppt+1]; break;
       case ADR:
          dectby( REALSIZE);
          dptr1 = (double *) &stk[current->ppt + 1];
          dptr2 = (double *) &stk[current->ppt - REALSIZE + 1];
          *dptr2 += *dptr1; break;
```

```
case SBI:
   dectby( 1); stk[current->ppt] -= stk[current->ppt+1]; break;
case SBR:
   dectby( REALSIZE);
   dptr1 = (double *) &stk[current->ppt + 1];
   dptr2 = (double *) &stk[current->ppt - REALSIZE + 1];
   *dptr2 -= *dptr1; break;
case MLI:
   dectby( 1); stk[current->ppt] *= stk[current->ppt+1]; break;
case MLR:
   dectby( REALSIZE);
   dptr1 = (double *) &stk[current->ppt + 1];
   dptr2 = (double *) &stk[current->ppt - REALSIZE + 1];
   *dptr2 *= *dptr1; break;
case DVI:
   dectby( 1); if (!stk[current->ppt+1]) ps = DIVCHK;
                 else stk[current->ppt] /= stk[current->ppt+1];
   break;
case DVR:
   dectby( REALSIZE);
   dptr1 = (double *) &stk[current->ppt + 1];
   dptr2 = (double *) &stk[current->ppt - REALSIZE + 1];
   if (!*dptr1)  ps = DIVCHK;
       else *dptr2 /= *dptr1;
   break;
case MOD:
   dectby( 1);
   if (!stk[current->ppt+1]) ps = DIVCHK;
       else if (stk[current->ppt+1] < 0) ps = RANCHK;
                else stk[current->ppt] %= stk[current->ppt+1];
   break;
case NOT:
   stk[current->ppt] = !stk[current->ppt]; break;
case NGI:
   stk[current->ppt] = -stk[current->ppt]; break;
case NGR:
   dptr1 = (double *) &stk[current->ppt - REALSIZE + 1];
   *dptr1 = -*dptr1;
   break;
case EQI:
   dectby( 1);
   stk[current->ppt] = stk[current->ppt+1] == stk[current->ppt];
   break;
case EQR:
   dectby( REALSIZE);
   dptr1 = (double *) &stk[current->ppt + 1];
   dptr2 = (double *) &stk[current->ppt - REALSIZE + 1];
   *dptr2 = *dptr2 == *dptr1;
   break;
case GTI:
   dectby( 1);
   stk[current->ppt] = stk[current->ppt] > stk[current->ppt+1];
   break;
case GTR:
   dectby( REALSIZE);
   dptr1 = (double *) &stk[current->ppt + 1];
   dptr2 = (double *) &stk[current->ppt - REALSIZE + 1];
   *dptr2 = *dptr2 > *dptr1; break;
```

```
case LSI:
   dectby( 1);
   stk[current->ppt] = stk[current->ppt] < stk[current->ppt+1];
   break;
case LSR:
   dectby(REALSIZE);
   dptr1 = (double *) &stk[current->ppt + 1];
   dptr2 = (double *) &stk[current->ppt - REALSIZE + 1];
   *dptr2 = *dptr2 < *dptr1;
   break;
case AND:
   dectby( 1);
   stk[current->ppt] = stk[current->ppt] && stk[current->ppt+1];
   break;
case ORR:
   dectby( 1);
   stk[current->ppt] = stk[current->ppt] || stk[current->ppt+1];
   break;
case FLO:
   dptr1 = (double *) &stk[current->ppt];
   *dptr1 = stk[current->ppt];
   incby(REALSIZE -1); break;
case FIX:
   dectby(REALSIZE - 1);
   dptr1 = (double *) &stk[current->ppt];
   stk[current->ppt] = *dptr1;
   break;
case MSI: /* accept a message */
   current->lastio = current->ppc - 1;
   size = 0;
   ioproc = partner(&size);  /* returns a procptr */
   i = 0; /* get message format */
   while ( (current->inout.format[i++] = nextcode()) >= 0
            && i < MAXMESS);
   if (ioproc == istd) /* std input */
        if (current->status == QUERY) guardcall(istd, INP);
          else readit(FALSE);
     else /* set up communications with ioproc */
      if (current->status == QUERY) grdcommun(ioproc, INP, size);
        else commun(ioproc,INP, size);
   break;
case MSO:
   current->lastio = current->ppc - 1;
   size = 0; ioproc = partner(&size);
   i = 0; /* get message format */
   while ( (current->inout.format[i++] = nextcode()) >= 0
            && i < MAXMESS) ;
   if (ioproc == NULL) writeit();
     else /* set up communications with ioproc */
       if (current->status == QUERY) grdcommun(ioproc, OUP,size);
          else commun(ioproc, OUP,size);
   break;
case STI:
   l = nextcode(); /* runtime declaration level */
   off = nextcode(); /* offset within frame */
   stk[current->display[current->lev-l] + off] = stk[current->ppt];
   dectby( 1);
   break;
```

```
case STR:
    l = nextcode();
    off = nextcode(); dectby(REALSIZE);
     /* value to be stored */
    dptr1 = (double *) &stk[current->ppt+1];
     /* destination address */
    dptr2 = (double *) &stk[current->display[current->lev-l]+off];
    *dptr2 = *dptr1;
    break;
case STA:
    size = nextcode(); /* size in stack units */
    l = nextcode(); /* runtime level */
    off = nextcode (); /* offset within frame */
    dectby(size);
    for (i=0; i < size; i++)
        stk[current->display[current->lev-l]+off+i] =
                        stk[current->ppt+1+i];
    break;
case SOI:
    off = stk[current->ppt];
    stk[off] = stk[current->ppt-1];
    dectby(2);
    break;
case SOR:
    off = stk[current->ppt]; dectby(REALSIZE+1);
    dptr1 = (double *) &stk[current->ppt+1]; /* value to be stored */
    dptr2 = (double *) &stk[off];
    *dptr2 = *dptr1;
    break;
case SOA:
    size = nextcode(); /* size in stack units */
    off = stk[current->ppt];
    dectby(size+1);
    for (i=0; i < size; i++)
          stk[off+i] = stk[current->ppt+1+i];
    break;
case LDI:
    l = nextcode();
    off = nextcode();
    incby(1);
    stk[current->ppt] = stk[current->display[current->lev-l] + off];
    break;
case LDR:
    l = nextcode(); off = nextcode();
    dptr1 = (double *) &stk[current->ppt+1]; /* destination */
    dptr2 = (double *) &stk[current->display[current->lev-l] + off];
    *dptr1 = *dptr2;
    incby(REALSIZE);
    break;
case LDA:
    size = nextcode(); /* size in stack units */
    l = nextcode(); /* runtime level */
    off = nextcode (); /* offset within frame */
    for (i=0; i< size; i++)
        stk[current->ppt+1+i] =
                stk[current->display[current->lev-l] + off + i];
    incby(size);
    break;
```

```
case LOI:
   off = stk[current->ppt];
   stk[current->ppt] = stk[off];  /* no inc or dec needed */
   break;
case LOR: /* load a real from the address at the top of the stack */
   dptr1 = (double *) &stk[stk[current->ppt]]; /* address */
   dptr2 = (double *) &stk[current->ppt];   /* destination */
   *dptr2 = *dptr1;
   incby(REALSIZE-1);
   break;
case LOA:
   size = nextcode(); /* size in stack units */
   off = stk[current->ppt];
   for (i=0; i< size; i++) stk[current->ppt+i] = stk[off + i];
   incby(size-1);  /* should have removed address */
   break;
case LDS: /* load string */ size = nextcode();
   for (i=0; i<size; i++) stk[current->ppt+1+i] = nextcode();
   incby(size);
   break;
case LSG: /* should not appear in a program that compiles */
   ps = CODCHK;  break;
case LCH:
   incby(1);
   if ( (stk[current->ppt] = nextcode()) ) IOSTD)
      {  /* swap process num to be on the top of the stack */
        stk[current->ppt+1] = stk[current->ppt];
        stk[current->ppt] = nextcode(); /* dim of process labels */
        incby(1); }
     else  i = nextcode(); /* do not need dimension for stdio */
   break;
case LID: /* Load the address onto tos */
   incby(1);
   l = nextcode();     /* relative level */
   off = nextcode();  /* variable offset */
   stk[current->ppt] = current->display[current->lev-l] + off;
   break;
case IND:
   l = nextcode();  /* lower dimension */
   i = nextcode();  /* upper dimension */
   off = nextcode(); /* size */
   if (stk[current->ppt] < l || stk[current->ppt] > i)
          error("Range error");
   if (l)  /* subtract lower bound dimension if not zero */
       stk[current->ppt] -= l;
   if (off != 1)   /* multiply by size of unit */
       stk[current->ppt] *= off;
   dectby(1); stk[current->ppt] += stk[current->ppt+1];
   break;
case PSI: incby(1); stk[current->ppt] = nextcode();  break;
case PSR:
   for (i = 0; i< REALSIZE; i++)
          stk[current->ppt+1+i] = nextcode();
   incby(REALSIZE); break;
case AJP:
   creatgrds(nextcode(),FALSE);  /* jump label */
   break;
```

```
case AJD:
   current->guardring->jumplab = nextcode();  break;
case FJP:
   if (! stk[current->ppt]) current->ppc = nextcode();
     else current->ppc++;
   dectby( 1); break;
case ALN: /* unconditional jump to start of jump table */
          current->status = QUERY;
case UJP: current->ppc = nextcode(); break;
case EDF:
    off = nextcode(); /* get new pc value */
    if (current->inrupt) /* found a partner communicant */
      { current->ppc = current->inrupt->jumplab;
        current->status = QUERY;
        current->guardlbl = current->guardring->grdun.dumnode.failadr;
        loadlabs(current, current->inrupt);
        current->inrupt = NULL;
      }
    else /* restart ALN command after removing all queries */
     { aptr oldhead;
       current->status = READY;
       current->ppc = off;
       while (oldhead = current->askhead)
        { remquery(current->askhead->askid, current, NULL);
          current->askhead = current->askhead->more;
          if (rlsmem ( (char *) oldhead, sizeof (struct asknode)) < 0)
             error("Incomplete memory release -- asknode");
        }
      }
     break;
case INC: incby(nextcode()); break;
case DEC: dectby(nextcode()); break;
case BEG: callproc(current); break;
case EXT: removeproc(current); break;
case HLT:
   ps = FINISHED; istd->status = globptr->status = TERMINATED;
   break;
case JBL:
   off = nextcode(); /* EBL address -- to check existing ring */
   if (!current->guardring ||
       current->guardring->grdun.dumnode.failadr != off)
         /* construct if ring doesn't exist -- for alt stmts
            in loops, ring not reconstructed every time */
         creatgrds(off,TRUE);
     else
        {   /* skip AJP codes */
          current->ppc = current->guardring->grdun.dumnode.failadr;
            /* reset fail values */
          current->curguard = current->guardring;
          while ( (current->curguard = current->curguard->gnext)
                    != current->guardring)
                 current->curguard->failed = FALSE;
        }
      /* this ensures that the guardring can never be tested first */
    current->curguard = current->guardring;
    current->guardring->grdun.dumnode.tested = 0;
    current->guardring->grdun.dumnode.failtest = 0;
    break;
```

```
    case EBL: /* actually starts execution of guards */
       current->guardlbl = -1; /* no longer interpreting guard */
       off = urand(current->guardring->grdun.dumnode.nodetot);
       for (i = 0; i < off; i++)
              current->curguard = current->curguard->gnext;
       testgrd(); break;
    case CRG: /* terminate alternative command successfully */
       swapgrings(); break;
    case DMP: dumpprocs(); break;
    case STK: dumpstack(current); break;
    case NOP: break;
    default:
       abort("Illegal opcode"); break;
  } /* end switch inst */
 }  /* nextstep */

/* ---------------------------------------------------------------------- */
```

```
/* Functions to interpret guarded commands, synchronization
    and communication */

#include "CSPIDEFS.C"
#include "CSPIEXT.C"

extern aptr newanode();
extern qptr newqrynode();
extern grdptr getguard();
extern int nextcode();
extern procptr search();
extern boolean inpstd;

/* ---------------------------------------------------------------------- */

 boolean labok(prl, lbls)
  int lbls[],prl[];
   /* Checks label equality of processes */

    { int i;

        for (i=0;i<MAXLAB,lbls[i] == prl[i];i++);
        return( (i==MAXLAB) ? TRUE : FALSE);
    }  /* labok */

/* ---------------------------------------------------------------------- */

 boolean formok(prl, pr2)
  int prl[], pr2[];
   /* Check that the message formats of two communicating processes
      are compatible */

   { int i = 0;

       while (prl[i] == pr2[i])
        { if (prl[i++] < 0) return(TRUE); }
       return(FALSE);  /* formats differ */
   }  /* formok */

/* ---------------------------------------------------------------------- */

 procptr partner(todec)
  int *todec;
   /* Finds process descriptor of communication process */

   { int labs[MAXLAB];
     int i,ioproc;
     procptr x;

       ioproc = stk[current->ppt];  /* remove procno */
       dectby(1);  (*todec)++;
       if (ioproc > IOSTD)
           { (*todec)++;   /* gives dimension */
             for (i=0; i<MAXLAB; i++) labs[i] = 0;
             for (i=stk[current->ppt--]; i>0; i--)
                  { labs[i-1] = stk[current->ppt--]; (*todec)++; }
           }
        else
```

```
        if (ioproc == INP) return(istd);    /* standard input */
         else return(NULL);                 /* standard output */
      /* Now find process descriptor of a labelled process */
    x = firstproc;
    while ( x = search(ioproc, x))
       if (labok(x->labs,labs)) break;
           else x = x->next;
    if (!x)  ps = IOCHK;  /* no descriptor found */
    return(x);
  }  /* partner */

/* ---------------------------------------------------------------------- */

 collapse(prr, inorout, form)
  procptr prr;          /* collapse stack of this process */
  boolean inorout;      /* input or output */
  int form[];           /* gives format of communication */
   /* Allows the removal of input addresses or output expressions from
      prr's stack, if partner process has terminated */

   { int comm, j = 0;

     while ( (comm = form[j++]) >= 0)
       if (comm == SGN || comm == MSS) {j++; continue; }
         else if (inorout == INP)  /* remove addresses */
                { prr->ppt--; j++; /* ignore size byte */ }
               else /* inorout == OUP; so remove expression */
                  prr->ppt -= form[j++];
   }  /* collapse */

/* ---------------------------------------------------------------------- */

 commun(pr, tag, toinc)
  procptr pr;
  short tag;      /* kind of communication done by current */
  int toinc;      /* number of stack spaces to restore if redoing inst */
   /* Sets up communication between the processes (pr) and (current) --
      assumes that current may suspend itself as it's not in a guard */

   { int i, comm, size, j = 0, incby(), loadlabs();
     qptr isquery(), found;

     if (pr->status == SUSP && pr->inout.mate == current &&
         pr->inout.direc == !tag &&
         formok(pr->inout.format,current->inout.format) )

        /* other process already waiting so proceed with transfer */
      { pr->status = READY; readyprocs++;  /* restart suspended process */
        while ( (comm = current->inout.format[j++]) >= 0)
             { size = current->inout.format[j++];
               if (comm == SGN || comm == MSS) continue;
               if (tag == INP) /* pr --> current */
                 { pr->ppt -= size; /* remove expr from pr's stack */
                   for (i=0; i<size; i++)
                     stk[stk[current->ppt]+i] = stk[pr->ppt +1 +i];
                   dectby(1); /* remove address from current's stack */
                 }
```

```
            else /* current --> pr */
             { dectby(size); /* remove expr from current's stack */
               for (i=0; i<size; i++)
                 stk[stk[pr->ppt]+i] = stk[current->ppt+1 +i];
               pr->ppt--; /* remove address from pr's stack */
              }
         } /* while */
    pr->inout.mate = NULL; /* not waiting any more */
  }
  else /* other half not ready for communication */
    if (pr->status == TERMINATED) /* fail -- collapse stack */
        { collapse(current,tag,current->inout.format);
          if (current->guardlbl >= 0) current->ppc = current->guardlbl;
        }
      else
        if (pr->status == QUERY)
          {  /* delay current by pr giving a non-committal answer --
                current keeps re-executing last MSO or MSI code */
            current->ppc = current->lastio; /* redo this i/o stmt */
            incby(toinc);           /* restore stack */
            current->status = WAIT;
               /* allow pr to get out of query mode */
            readyprocs--; current->inout.mate = pr;
            return;
          }
         else
          if (found = isquery(pr, tag))
           {  if (pr->status == IDLE)
            /* current has been queried by pr before and pr is waiting --
                so suspend and allow pr to complete communication */
            {   /* commence execution at i/o guard */
              pr->ppc = found->querypc->jumplab;
              pr->status = QUERY; loadlabs(pr, found->querypc);
              pr->guardlbl = pr->guardring->grdun.dumnode.failadr;
              current->status = SUSP; /* readyprocs stays constant */
              current->inout.mate = pr;
              current->inout.direc = tag; return;
            } /* isquery && IDLE */

             else if (pr->status == USELESS)
               /* current has been queried by pr before and pr is
                   executing the otherwise guard -- so suspend and
                   allow pr to complete communication */
            { pr->inrupt = found->querypc;
              current->status = SUSP; readyprocs--;
              current->inout.mate = pr;
              current->inout.direc = tag; return;
             } /* isquery && USELESS */
            }
             else
              /* suspend current process and fill in inout link */
              { current->status = SUSP; readyprocs--;
                current->inout.mate = pr;
                current->inout.direc = tag;
               }
} /* commun */
```

```
/* ----------------------------------------------------------------------- */

 makequery (pr, tag)
  procptr pr;
  short tag;       /* kind of communication done by process pr */
   /* Sets up a new query node in the process descriptor of pr -- also
      adds a queried (ask) node in process descriptor of current */

   { qptr qr; aptr ask;
     register i = -1;

     qr = newqrynode(); qr->more = pr->queryhead;
     pr->queryhead = qr; qr->queryid = current;
     qr->querydirect = tag;
     qr->querypc = current->curguard;
     do  i++;
       while ( (qr->qformat[i] = current->inout.format[i]) >= 0);
     ask = newanode();  ask->more = current->askhead;
     current->askhead = ask;  ask->askid = pr;
   }  /* makequery */

/* ----------------------------------------------------------------------- */

 grdcommun(pr, tag, toinc)
  procptr pr;
  short tag;      /* kind of communication done by current */
  int toinc;      /* number of stack spaces to restore if redoing inst */
   /* Sets up communication between the processes (pr) and (current) --
      assumes that current may not suspend itself as it is in a guard */

   { int i, comm, size, j = 0, remquery(), incby(),loadlabs();
     aptr oldhead;
     qptr found,isquery();

     if (pr->status == SUSP && pr->inout.mate == current &&
         pr->inout.direc == !tag &&
         formok(pr->inout.format,current->inout.format) )

      /* other process already waiting so proceed with transfer - even
         if input/output command appears in a guard */
       { pr->status = READY; readyprocs++;  /* restart suspended process */
         while ( (comm = current->inout.format[j++]) >= 0)
           { size = current->inout.format[j++];
             if (comm == SGN || comm == MSS) continue;
             if (tag == INP) /* pr --> current */
               { pr->ppt -= size; /* remove expr from pr's stack */
                 for (i=0; i<size; i++)
                    stk[stk[current->ppt]+i] = stk[pr->ppt +1 +i];
                 dectby(1); /* remove address from current's stack */
               }
              else /* current --> pr */
                { dectby(size); /* remove expr from current's stack */
                  for (i=0; i<size; i++)
                    stk[stk[pr->ppt]+i] = stk[current->ppt+1 +i];
                  pr->ppt--; /* remove address from pr's stack */
                }
           } /* while */
```

```
    while (oldhead = current->askhead)
      /* i.e process had polled others */
        { remquery(current->askhead->askid, current, NULL);
          current->askhead = current->askhead->more;
          if (rlsmem ( (char *) oldhead, sizeof (struct asknode)) < 0)
              error("Incomplete memory release -- asknode");
        }
    pr->inout.mate = NULL; /* not waiting any more */
    return;
  }
else
   if (found = isquery(pr, tag))
    { if (pr->status == IDLE)
      /* current has been queried by pr before and pr is waiting --
         so suspend and allow pr to complete communication */
      {  /* commence execution at i/o guard */
        pr->ppc = found->querypc->jumplab;
        pr->status = QUERY;  loadlabs(pr, found->querypc);
        pr->guardlbl = pr->guardring->grdun.dumnode.failadr;
        current->status = SUSP; /* readyprocs stays constant */
        current->inout.mate = pr;
        current->inout.direc = tag;
        while (oldhead = current->askhead)
            /* i.e process had polled others */
          { remquery(current->askhead->askid, current, NULL);
            current->askhead = current->askhead->more;
            if (rlsmem ( (char *) oldhead, sizeof (struct asknode)) < 0)
                error("Incomplete memory release -- asknode");
          }
        return;
       }
       else if (pr->status == USELESS)
      /* current has been queried by pr before and pr is currently
         executing the default guard -- so suspend and allow pr
         to complete communication */
      { pr->inrupt = found->querypc;
        current->status = SUSP;  readyprocs--;
        current->inout.mate = pr;
        current->inout.direc = tag;
        while (oldhead = current->askhead)
            /* i.e process had polled others */
          { remquery(current->askhead->askid, current, NULL);
            current->askhead = current->askhead->more;
            if (rlsmem ( (char *) oldhead, sizeof (struct asknode)) < 0)
                error("Incomplete memory release -- asknode");
          }
        return;
       }
       else
        if (pr->status == QUERY)
          if (current->unid > pr->unid)
           {  /* delay curguard by pr giving a non-committal answer --
                   current carries on executing other guards and when all
                   others have been tested retests this one */
              current->ppc = current->guardlbl; /* ignore last inst */
              current->curguard->failed = FALSE;
              current->guardring->grdun.dumnode.failtest--;
              current->guardring->grdun.dumnode.tested--;
```

```
                collapse(current,tag, current->inout.format);
                return;
              }
            else /* current->unid <= pr->unid */
              { /* pr has queried current - now reverse this action */
                  /* guard in pr reset to status as if waiting */
                found->querypc->failed = FALSE;
                found->queryid->guardring->grdun.dumnode.tested--;
                  /* delete entry 'found' in current's query queue */
                remquery(current, NULL, found);
                  /* delete entry in pr's asked queue */
                remask(pr, current);
                  /* now do makequery part below -- no return yet */
              }
        } /* else isquery */

     /* other half not ready for communication or busy with own queries;
        input/output is in guarded statement, so guard fails only if process
        partner has terminated - otherwise guard assumes pending state */
     current->ppc = current->guardlbl; /* jump to fail address of guard */
     if (pr->status != TERMINATED)
         { current->guardring->grdun.dumnode.failtest--;
           current->curguard->failed = PENDING; /* does not fail */
           makequery(pr, !tag);  /* current wants a !tag from pr */
         }
     collapse(current,tag,current->inout.format);
   } /* grdcommun */

/* ---------------------------------------------------------------------- */

 qptr isquery(prr, direc)
  procptr prr;
  short direc;
   /* Checks whether current process has been queried by prr before */

  { qptr temp = current->queryhead;
    while (temp && (temp->queryid != prr ||  temp->querydirect != direc ||
                    !formok(current->inout.format, temp->qformat) ) )
       temp = temp->more;
    return(temp);
  }  /* isquery */

/* ---------------------------------------------------------------------- */

 remquery(prr, id, qid)
  procptr prr, id;
  qptr qid;
   /* Removes a query by id from prr's query list once id has accepted
      communication with a process */

   { qptr prev = NULL,
          found = prr->queryhead;

     while (found && ((id && found->queryid != id) || (!id && found != qid)))
       { prev = found; found = found->more; }
     if (found)
        { if (prev) prev->more = found->more;
            else prr->queryhead = found->more;
```

```
        if (rlsmem( (char *) found, sizeof (struct qrynode)) < 0)
            error ("Incomplete memory release -- qrynode");
      }
  }  /* remquery */

/* ---------------------------------------------------------------------- */

remask(pr, id)
 procptr pr, id;
  /* Removes the query by pr to id in pr's asked list, in order to
     reverse the querying process */

  { aptr prev = NULL,
         found = pr->askhead;

    while (found && found->askid != id)
       { prev = found; found = found->more; }
    if (found)
       { if (prev) prev->more = found->more;
           else pr->askhead = found->more;
         if (rlsmem ( (char *) found, sizeof (struct asknode)) < 0)
           error("Incomplete memory release -- asknode");
       }
  }  /* remask */

/* ---------------------------------------------------------------------- */

loadlabs(prs, grd)
 grdptr grd;
 procptr prs;
  /* Load lab values of restarted guard */

  { register i;

      if (grd->grdun.altnode.dm)   /* dim > 0 */
         for (i = 0; i<grd->grdun.altnode.dm; i++)
            { stk[++prs->ppt] = grd->grdun.altnode.glabs[i]; }
  }  /* loadlabs */

/* ---------------------------------------------------------------------- */

readit(sigged)
 boolean sigged;   /* whether to read signal or not */
  /* Does input from keyboard depending on format */

 { int chk,comm, j = 0;
   char sig;
   double *dptrl;

    while ((comm = current->inout.format[j++]) >= 0 && ps == RUNNING)
     { chk = 1;
       switch (comm)
       { case ITG: if (inpstd) putchar(':');
                   chk = fscanf(dfile,"%d%*c", &stk[stk[current->ppt]]) - 1;
                   dectby(1); break;
         case CHR: if (inpstd) putchar(':');
                   stk[stk[current->ppt]] = getc(dfile);
                   dectby(1); break;
```

```
        case REL: /* input a real and store in address on tos */
                  if (inpstd) putchar(':');
                  dptrl = (double *) &stk[stk[current->ppt]];
                  chk = fscanf(dfile, "%lf%*c", dptrl) - 1;
                  dectby(1);  /* remove address */
                  break;
        case SGN: /* read anything for signal - do not store */
                  if (!sigged) { if (inpstd) putchar(':');
                                 getch(); }
                  break;
        case MSS: /* only used to match message formats */
                  break;
        default: /* no array input allowed */
                  dectby(1);  /* remove address */
                  error("Array input not implemented");
                  break;
      }
     j ++;  /* don't use size byte for std input */
     if (chk == EOF ) ps = EOFCHK;
       else if (chk != 1) ps = IOCHK;
    } /* while */
  }  /* readit */

/* ---------------------------------------------------------------------- */

 writeit()
  /* Does output to screen depending on format */

  { int j = 0, comm;
    double *dptrl;

    if (tracing) fprintf(rfile, "%40s", "-------> ");
    while ((comm = current->inout.format[j++]) >= 0)
     { switch (comm)
       { case ITG: fprintf(rfile,"%d ", stk[current->ppt]);
                   break;
         case CHR: fprintf(rfile, "%c", stk[current->ppt]);
                   break;
         case REL:
                dptrl = (double *) &stk[current->ppt - REALSIZE + 1];
                fprintf(rfile, "%lf ", *dptrl);
                break;
         case SGN: /* output anything for signal */
                   fprintf(rfile, "SG ");
                   j++;  /* move past size/number byte */
                   break;
         case MSS: j++; break;  /* only used to match message formats */
         default: /* no array output allowed */
                  error("Array output not implemented");
                  break;
       }
      if (comm != SGN && comm != MSS)
        dectby(current->inout.format[j++]); /* decrease by size */
    } /* while */
  if (tracing) putc('\n', rfile);
  }  /* writeit */
```

```
/* ----------------------------------------------------------------------- */

testgrd()
 /* Test the guard ring -- if untested guard found, transfer to it;
    else pc counter drops down to next instruction after guard table */

  { int i, dm, urand();

    if (current->guardring->grdun.dumnode.tested++ <
              current->guardring->grdun.dumnode.nodetot)
      /* still a guard untested */
        { do { current->curguard = current->curguard->gnext; }
              /* skip failed nodes,  noncommital nodes and dummy */
            while (current->curguard->failed);
            /* load lab values if there */
          if (current->curguard->grdun.altnode.dm)  /* dim > 0 */
            for (i = 0; i<current->curguard->grdun.altnode.dm; i++)
             { incby(1);
               stk[current->ppt] = current->curguard->grdun.altnode.glabs[i];
             }
              /* assume pending state will not be reached */
            current->curguard->failed = TRUE;
            current->guardring->grdun.dumnode.failtest++;
            current->ppc = current->curguard->jumplab; /* transfer control */
        }
      else
        if ( (--current->guardring->grdun.dumnode.tested) !=
                current->guardring->grdun.dumnode.failtest)
             { /* not all guards failed -- so process becomes idle, OR
                  if OTHERWISE guard present, this is executed */
               current->guardring->grdun.dumnode.failtest,
               current->guardring->grdun.dumnode.tested);
               if (current->guardring->jumplab >= 0)
                   /* do otherwise guard */
                 { current->ppc = current->guardring->jumplab;
                   current->status = USELESS;
                   sigwaits(current); }
                else
                 { current->status = IDLE;  readyprocs--;
                   sigwaits(current);
                   current->ppc = current->guardring->grdun.dumnode.failadr;
                 }  /* else */
             }
        else   /* all guards false -- alternative command fails */
         {  current->status = READY; /* no longer in query mode */
            sigwaits(current); swapgrings(); }
   }  /* testgrd */

/* ----------------------------------------------------------------------- */

sigwaits(thispr)
 procptr thispr;
  /* Find any other processes waiting on thispr and reactivate */

  { procptr x = firstproc;
     do
      { if (x->status == WAIT && x->inout.mate == thispr)
          { x->status = READY; x->inout.mate = NULL; readyprocs++; }
```

```
        x = x->next;
      } while (x != firstproc);
  }  /* sigwaits */

/* ---------------------------------------------------------------------- */

swapgrings()
 /* Reinstate outer guard ring once nested alternative command has
    terminated -- reclaim storage used by inner ring */

  { grdptr rel, start, temp;
     start = current->guardring; temp = start->gnext;
     current->curguard = current->guardring->grdun.dumnode.prevcur;
     current->guardring = current->guardring->grdun.dumnode.prevhead;
       /* reclaim space occupied by guard nodes */
     while ( (rel = temp) != start )
       { temp = temp->gnext;
         if (rlsmem( (char *) rel, sizeof (struct grdnode)) < 0)
             error("Incomplete memory release");
       }
     if (rlsmem( (char *) start, sizeof (struct grdnode)) < 0)
            error("Incomplete memory release");
  }  /* swapgrings */

/* ---------------------------------------------------------------------- */

creatgrds(jmplab,dummy)
 int jmplab;
 boolean dummy;
  /* Create a new guard node and insert into ring just after dummy node */

  { int i;
    grdptr newgard;

    newgard = getguard();
    if (dummy)
     { newgard->jumplab = -1;
       newgard->grdun.dumnode.failadr = jmplab;
       newgard->failed = TRUE;  /* prevents dummy node being tested */
       newgard->grdun.dumnode.nodetot = 0;
       newgard->grdun.dumnode.prevhead = current->guardring;
       newgard->grdun.dumnode.tested = newgard->grdun.dumnode.failtest = 0;
       newgard->grdun.dumnode.prevcur = current->curguard;
       current->guardring = newgard; current->guardring->gnext = newgard;
     }
    else  /* alternative guard node */
      { newgard->failed = FALSE;
        newgard->jumplab = jmplab;
        newgard->grdun.altnode.dm = nextcode();
        for (i=0; i<newgard->grdun.altnode.dm; i++)
              newgard->grdun.altnode.glabs[i] = nextcode();
        newgard->gnext = current->guardring->gnext;
        current->guardring->gnext = newgard;
        current->guardring->grdun.dumnode.nodetot++;
      }
   }  /* creatgrds */

/* ---------------------------------------------------------------------- */
```

```
/* Functions to interpret standard input guards */

#include "CSPIDEFS.C"
#include "CSPIEXT.C"

#define BUFMAX 100

static int elements = 0;

/* ----------------------------------------------------------------------- */

 static qptr findrequest(prev)
  qptr prev;
   /* Find a user process waiting for input -- uses FIFO queue */

  { qptr found = NULL, next;

    next = istd->queryhead;
    while (next != NULL && next != prev)
       { /* find first request waiting */
         found = next;
         next = next->more; }
    if (found && found->queryid->status != SUSP &&
        found->queryid->status != IDLE && found->queryid->status != USELESS)
           return( findrequest(found) );
      else return(found);
   }  /* findrequest */

/* ----------------------------------------------------------------------- */

 fudgeinput()
  /* Allows the standard input process to receive a timeslice.  In
     addition handles requests from user processes for input. */

  { qptr found;

    while ( kbhit() )
     { if (elements++ > BUFMAX)  /* overflow */ break;
       getch();  }
    if (istd->queryhead != NULL && elements > 0)
       /* process a request from a user process */
     { found = findrequest(NULL);
       if (found) stdanswer(found->queryid, found);
       current = istd;
     }
  }  /* fudgeinput */

/* ----------------------------------------------------------------------- */

 stdanswer(pr,found)
  procptr pr;
  qptr found;
   /* Reply to request from user process (pr) for input and polls
      the keyboard to pick up any user intervention. */

 { if (pr->status == SUSP && pr->inout.mate == istd)
     /* pr already waiting so proceed with the read */
    { pr->status = READY; readyprocs++; /* restart suspended process */
```

```
      pr->inout.mate = NULL;
      remquery(current, NULL, found);
      remask(pr, current);
      elements--; current = pr; readit(TRUE);
      return;
     }
   else  /* pr not suspended */
     if (pr->status == IDLE)
        /* suspend istd and let pr complete the communication */
       { pr->ppc = found->querypc->jumplab;
         pr->status = QUERY; loadlabs(pr, found->querypc);
         pr->guardlbl = pr->guardring->grdun.dumnode.failadr;
         current->status = SUSP; /* readyprocs stays constant */
         current->inout.mate = pr;
         current->inout.direc = OUP;
       }
       else if (pr->status == USELESS)
        /* suspend istd and let pr complete the communication after
           executing the default guard */
       { pr->inrupt = found->querypc;
         current->status = SUSP; readyprocs--;
         current->inout.mate = pr;
         current->inout.direc = OUP;
       }
   }  /* stdanswer */

/* ---------------------------------------------------------------------- */

 guardcall(pr, tag)
  short tag;
  procptr pr;
   /* Executed if user process wishes to communicate with the standard
      input device (pr) within a guarded command */

 {  aptr oldhead;

    if (pr->status == SUSP && pr->inout.mate == current &&
        pr->inout.direc == !tag)
     { pr->status = READY; readyprocs++; /* restart std input */
       pr->inout.mate = NULL;
       while (oldhead = current->askhead)
         /* process had polled others */
       { remquery(current->askhead->askid, current, NULL);
         current->askhead = current->askhead->more;
         if (rlsmem ( (char *) oldhead, sizeof (struct asknode)) < 0)
            error ("Incomplete memory release -- asknode ");
       }
       elements --; readit(TRUE);
      }
     else  { current->ppc = current->guardlbl;
             current->curguard->failed = PENDING;
             current->guardring->grdun.dumnode.failtest--;
             makequery(pr, !tag);
             collapse(current, tag, current->inout.format);
            }
   }  /* guardcall */

/* ---------------------------------------------------------------------- */
```