University of Cape Town
Department of Computer Science

An Estelle Compiler

by
Jacques van Dijk

A Thesis
Prepared under the Supervision of
Prof. P.S. Kritzinger
In fulfilment of the Requirements for the
Degree of Master of Science in Computer Science.

Cape Town
June, 1988.

Acknowledgements

I would like to thank:

My Data Network Architectures Laboratory collegues, Janette Punt, Johann Dreyer, Graham Wheeler and Pieter Kritzinger for many helpful discussions concerning Estelle and the Estelle compiler system.

Pieter de Villiers and Willem van Biljon who were a source of information and solid advice concerning compilers and programming languages.

Mark Meumann for help with the mysteries of LaTeX and Marietjie Steyn for the artwork.

My supervisor, Pieter Kritzinger, who made the project a valuable experience.

My parents and friends for their continued support.

The CSIR who supported my studies with bursaries.

Contents

Abstract

The increasing development and use of computer networks has necessitated international standards to be defined. Central to the standardization efforts is the concept of a Formal Description Technique (FDT) which is used to provide a definition medium for communication protocols and services. This document describes the design and implementation of one of the few existing compilers for the one such FDT, the language "Estelle" ([ISO85], [ISO86], [ISO87]).

Chapter 1

Introduction.

This thesis describes the design and implementation of a compiler for the language "Estelle", ([ISO85], [ISO86], [ISO87]) a Formal Description Technique (FDT) for communication protocols.

The compiler was designed as part of a Protocol Development Workbench [Kri87] based on Estelle. The workbench will provide a protocol designer, or implementor, with a system for protocol validation and performance prediction, as well as aiding with implementation testing. (A similar system is described for FAPL [Poz82].)

Although the development of a compiler is no longer considered to be as difficult as was the case in the 1950's [Aho86], it still represents a substantial expenditure of time and effort. The implementation of yet another language necessitates a careful appreciation of the issues. The first phase of this compiler project was, therefore, an investigation of the properties necessary for implementing communication software architectures, the extent to which various languages (Ada, C, Edison, Estelle and Modula-2) satisfy these requirements and finally the advisibility of implementing Estelle as opposed to the use of an existing language as a standard implementation language [Kri86]. This investigation also questioned the use of an existing high-level target language for an Estelle compiler.

The second phase was the development of an LL(1)-based parser for a subset of Estelle as defined in the first draft proposal [ISO85], and the associated tools needed for LL(1) table generation [vDi87]. This served as familiarization with compilation, as the development team had no previous experience in this field.

The third phase was the development of a complete compiler for Estelle. This had three distinct development steps: a compiler for a Pascal subset

[vDi87a], a compiler for the entire Pascal subset contained in Estelle, and a compiler for the entire Estelle.

The compiler implemented is one of the few existing Estelle compilers, the general literature citing only three previous examples, namely by the U.S. National Bureau of Standards [Lin86], Ansart et al. [Ans87] and de Souza and Ferneda [dSo88]. In addition, de Saqui-Sanni and Courtiat [dSa87] reported the development of an Estelle interpreter.

This thesis consists of background to Estelle and compilation, a design overview and an implementation description.

In the next chapter Estelle is described and discussed.

Chapter 2

Estelle.

As a result of the increasing development and use of computer networks in the mid-1970's, it became apparent that, to ensure the full exploitation of this rapidly emerging technology, international standards were necessary. In 1978 this resulted in the formation of a new subcommittee (SC16) of the International Organization for Standards (ISO) Technical Committee 97 on Information Processing. The new subcommittee for "Open Systems Interconnection" would provide standards which would make a system conforming to the standard "open" to communication with any other system obeying the same standards anywhere in the world [Day83].

ISO/TC97/SC16's priority was the development of an architecture for open systems interconnection which would serve as a framework in which standard protocols could be defined [Zim80]. This goal was achieved with the definition of OSI Reference Model [Zim80], [Day83]. To satisfy the goals of OSI Formal Description Techniques (FDT's) were needed to define protocols unambiguously so that implementors anywhere in the world could develop correct and compatible implementations. Working Group 1 (WG1) of SC 16 therefore formed a separate group on FDT's [Vis83].

The FDT group later formed three subgroups: A) dealing with architectural specification concepts, B) dealing with the development of a FDT based on the Extended Finite State Machine (FSM) concept and C) dealing with the development of a FDT based on the Temporal Ordering of Interaction Primitives [Vis83]. Subgroups B and C lead respectively to the development of the Extended State Transition Language or Estelle ([ISO85], [ISO86], [ISO87]) and the Language for Temporal Ordering Specification or Lotos [Vis83].

The Extended Finite State Machine concept is discussed next.

2.1 The Extended Finite State Machine Model.

Formal Description Techniques used to specify protocols include state machine specifications, Petri-net specifications, programming language descriptions, Lamport's concurrent module approach, event specification and temporal logic-based approaches [Sch82].

The various specification methods are discussed in more detail in [Pet77] (Petri-nets), [Boc80] (State transitions - programming languages), [Sch82] (State Machines, Petri-nets, programming languages, Lamport's concurrent module approach, sequence expression languages, event specifications and state-based temporal logic approaches) and [Dan80] (Finite state automata, Petri-nets).

The Extended Finite State Machine model is a combination of aspects of finite state transition diagrams and programming languages [Boc80]. A finite state transition diagram and program variables are used to define the possible states of the entity. The advantage of this method of specification is that most protocols contain aspects that lend themselves to description by finite state transition diagrams, while other protocols can more elegantly be described via program fragments, i.e. variables and statements. In addition, the OSI's layered architecture and modularization can elegantly be described with programming language concepts.

In the classical definition, a finite state transition diagram is a directed graph in which the nodes and edges represent respectively states and transitions of a finite state automaton (FSA). A FSA is denoted by a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ a finite set of input symbols, q_0 an initial state, $F \subset Q$ a set of final states and δ is the transition function which maps $Q \times \Sigma \rightarrow Q$. That is, for every $q \epsilon Q$ and every $a \epsilon \Sigma$, $\delta(q, a)$ is a state. These classical FSA's are used to recognize strings in a language $L(M)$ defined by the machine M. Finite automata which allow output apart from the acceptance/non- acceptance of a string are either Moore or Mealy machines. (Moore if the output is associated with a state and Mealy if associated with the transitions.) These machines are defined as a 6-tuple $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where Q, Σ, δ, and q_0 are as defined before, and Δ is the output alphabet. λ, the output function, maps Q to Δ in the case of Moore machines and $Q \times \Sigma$ to Δ for Mealy machines.

For example, Figure 2.1. is the graphic representation of the Mealy machine $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where $Q = \{A, B, C\}, \Sigma = \{0, 1\}, \Delta = \{a, b, c\}, q_0 = A$ and δ and λ are defined as:

$\delta(A, 0) = B$, $\lambda(A, 0) = b$,

$\delta(A,1) = C$, $\lambda(A,1) = c$,
$\delta(B,0) = A$, $\lambda(B,0) = a$,
$\delta(B,1) = A$, $\lambda(B,1) = a$,
$\delta(C,0) = A$, $\lambda(C,0) = a$,
$\delta(C,1) = A$, $\lambda(C,1) = a$.

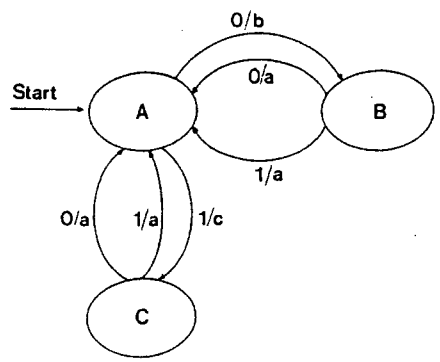

Figure 2.1: Graphic Representation of a Mealy Machine.

Although the definition of finite state machines is less rigorous in the data communication field, the FSM's used in the specification of communications can be classified as extended Mealy machines. The extensions used usually concern the use of program variables and additional conditions for transitions. Examples of these extensions will be discussed in the section concerning transitions in Estelle.

The next section is an introduction to Estelle as defined in the DIS 9074 [ISO87].

2.2 Overview of Estelle.

Estelle can be characterized as an extension of a subset of Pascal, which contains the module concept, where each of the nested modules is specified as an extended finite state machine.

A module consists of two parts, a header and one or more body-parts. The module header describes the external visibility of a module instance. This visibility is described in terms of a formal parameter list used to pass parameters for module initialization, exported variables which are shared between parent and child modules, and interaction points which define full

duplex interfaces with interaction points of another module via channels. Note that the interaction points, exported variables and formal parameter lists are optional.

A module body defines the utility of a module. By associating a module header with different module bodies, module instances can be dynamically created which exhibit different behaviour, although having the same interface. A module body consists of three parts: the declaration-part, the initialization-part and the transition-part, all of which are optional.

The declaration-part contains parts for the definition of constants, types, variables, procedure and functions, as well as parts for the definition of constructs unique to Estelle such as channels, module headers, module bodies, interaction points, module variables, states and state sets.

The initialization-part specifies code to be executed immediately after module instance creation. This includes code to set the initial state of the FSM and to connect it's interaction points to external or internal interaction points.

The transition-part is used to specify the FSM of the module. The input of the conceptual FSM are the events received through channels from other modules.

The definition of a FSM for a module in Estelle is as follows: (The example given in section 2.1. is used once more, with "$Event_0$", "$Event_1$", "$Event_a$", "$Event_b$" and "$Event_c$" replacing the input and output symbols "0" and "1", and "a", "b" and "c", respectively)

```
(* The following is the declaration-part of the module: *)

state A, B, C;
stateset B_or_C = [B,C];

(* The initialization-part of the module: *)

initialize
to A
begin
   (* Code for initialization of the module's internal
      variables and interaction points. etc *)
end;

(* The following is the transition-part of the module: *)
```

```
trans
from A
  when Event0 to B
    begin output Eventb end (* Transition 1 *)

  when Event1 to C
    begin output Eventc end (* Transition 2 *)

from B
  when Event0 to A
    begin output Eventa end ( Transition 3 *)

  when Event1 to A
    begin output Eventa end (* Transition 4 *)

from C
  when Event0 to A
    begin output Eventa end (* Transition 5 *)

  when Event1 to A
    begin output Eventa end (* Transition 6 *)

end; (* trans *)
```

Note the nesting of the clauses before transitions; "**from** *A*" holds for both the "**when** $Event_0$" and the "$Event_1$". Note also that the last four transitions can group together as follows:

```
from B_or_C
  when (Event0 or Event1) to A
    begin output Eventa end ( * Transition 3 *)
```

2.2.1 Clauses.

The from-clause specifies the current state, the when-clause the event (input for a classical FSA), the to-clause the next state. Further clauses can be placed to extend the transition conditions. For example, the provided-clause specifies a boolean guard to the transition, the delay-clause allows spontaneous transitions, i.e. transitions which contain no when-clause, to be

executed only after a time-interval specified in the clause, while the priority-clause allows transitions to be placed in various orders of priority. The any-clause is used to allow shorthand specification of a number of transitions.

The when-clause of a transition allows input from another module, while the output-statement allows output. In Estelle this input/output is called an interaction, and the interaction is qualified to indicate with which interaction point it is associated, and therefore which channel and which interaction points of other modules.

2.2.2 Interaction Points.

The interaction points define abstract interfaces through which interactions can be exchanged between modules. Each interaction point has three attributes: queuing discipline, channel type and role. The queuing discipline determines whether the interaction point is associated with a individual queue, or if it shares a queue with other interaction points. The channel type associates a set of permissible interactions with the interaction point. A channel type has two sets of interactions, (not necessarily disjoint), each associated with a role. The role of a interaction point identifies the set of permissible output interactions, as well as the permissible input interactions. Interactions in a set of interactions which match the role are permissible for output, while the other set is permissible for input.

In addition to the above attributes an interaction point has the attribute internal or external. Interaction points defined in a module header are external, while those defined in the declaration-part of a module body are internal. This attribute defines the visibility of an interaction point to module instances. An external interaction point of children are visible to their parent modules, while their internal interaction points are not. Interactions may be exchanged between the internal interaction points of a parent module and the external interaction points of its children.

2.2.3 Interactions.

An interaction may be some form of user-defined data type representing an event in the module specification. Interactions may be messages consisting of structured data to be exchanged between modules. Protocol data units can also be represented as interactions. These user-defined data types are first specified and then enumerated as permissible interactions over a channel type.

2.2.4 Channels.

Channel types are used to associate a channel type identifier with a number of interactions. A role identifier allows the identification of a set of interactions. Each channel is associated with two roles and each interaction is either associated with one or both of the roles.

The following example demonstrates the use of channels, interactions and interaction points to specify the classic readers/writers problem: (Figure 2.2.)

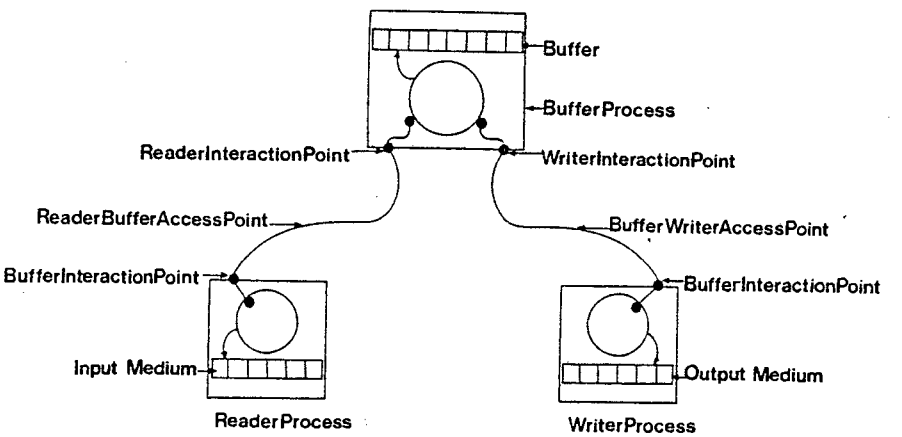

Figure 2.2: Graphic Representation of the Readers/Writers Problem.

A process called "*ReaderProcess*" reads data from an input medium and then writes it to a buffer, where another process, "*WriterProcess*", reads it from the buffer and writes it to an output medium. When the buffer is full, "*ReaderProcess*" must suspend writing to the buffer, and likewise, when the buffer is empty, "*WriterProcess*" must suspend reading from the buffer. This can be defined by the following:

channel *ReaderBufferAccessPoint*(*Reader*, *Buffer*);
  by *Reader*: (* Define which interactions may be output *)
  (* when an interaction point is associated *)
  (* with the role *Reader* *)

  WriteToBuffer(*Message*: *MessageType*);

  by *Buffer*: (* Define which interactions may be output *)

```
            (* when an interaction point is associated *)
            (* with the role Buffer *)

    BufferFull;
    WriteConfirmed;

channel BufferWriterAccessPoint(Buffer, Writer);
  by Buffer: (* Define which interactions may be output *)
            (* when an interaction point is associated *)
            (* with the role Buffer *)

    RequestedMessage(var Message: MessageType);
    BufferEmpty;

  by Writer: (* Define which interactions may be output *)
            (* when an interaction point is associated *)
            (* with the role Writer *)

    RequestData;

module ReaderProcess process;
  ip BufferInteractionPoint: ReaderBufferAccessPoint(Reader)
     individual queue;
end;

module WriterProcess process;
  ip BufferInteractionPoint: BufferWriterAccessPoint(Writer)
     individual queue;
end;

module BufferProcess process;
  ip ReaderInteractionPoint: BufferWriterAccessPoint(Buffer)
     individual queue;
     WriterInteractionPoint: BufferWriterAccessPoint(Buffer)
     individual queue;
end;
```

"*ReaderBufferAccessPoint*" defines a channel between two modules associated with the roles "*Reader*" and "*Buffer*". The interaction

"*WriteToBuffer*" may be output by a process with the role "*Reader*", while the interactions "*BufferFull*" and "*WriteConfirmed*" are available to a process associated with the role "*Buffer*". In the same way "*BufferWriterAccessPoint*" defines a channel between two modules associated with the roles "*Buffer*" and "*Writer*". The interactions "*BufferEmpty*" and "*RequestedMessage*" may be output by a process with the role "*Buffer*", while the interaction "*RequestData*" is available to a process associated with the role "*Writer*". "*ReaderProcess*", "*WriterProcess*" and "*BufferProcess*" are module headers respectively defining a process with a "*ReaderBufferAccessPoint*" channel type with the role "*Reader*", a process with a "*BufferWriterAccessPoint*" channel type with the role "*Writer*", and a process with a "*ReaderBufferAccessPoint*" and a "*BufferWriterAccessPoint*" channel type both associated with the role identifier "*Buffer*". Note that this identifier need not be the same for both channel types.

The readers/writers problem is now reduced to an interaction between the three processes "*ReaderProcess*", "*WriterProcess*" and "*BufferProcess*". When "*ReaderProcess*" has read data from the input medium, it performs the instruction
"**output** *BufferInteractionPoint.WriteToBuffer*(*Message*)"
which defines that the interaction must be sent to the interaction point "*BufferInteractionPoint*". The process then goes into a state which has transitions with the following two when-clauses:
"**when** *BufferInteractionPoint.BufferFull*" and
"**when** *BufferInteractionPoint.WriteConfirm*".
This will allow for either retransmission attempt, or the reading of the following data from the input medium. Concurrently with "*ReaderProcess*" "*WriterProcess*" sends the interaction "*RequestData*" to "*BufferProcess*" via its "*BufferInteractionPoint*" and waits for either the interaction "*BufferEmpty*" or the interaction "*RequestedData*" to be sent from "*BufferProcess*". This allows it to either rerequest data or write the data to the output medium.

2.2.5 Module Instance Creation and Release.

A module instance is created and initialized with the init-statement of Estelle. A module variable is associated with a module header and a module body to define a module instance. The actual parameter list given in the init-statement is used to initialize the values associated with the correspond-

ing formal parameter list of the module header. The initialization part of the module body is then executed to place the FSM in its initial state. If no initialization part exists for a module body, the module instance is in what is called a "preinitial" state. After the init-statement the module variable identifies this module instance.

The release-statement of Estelle is used to release module instances at a lower level in the module hierarchy. All external interaction points of the module instance identified by the module-variable which have been attached or connected are detached or disconnected, and the module instance and all its descendant instances are released and are no longer available. The value of all module-variables which identified released module instances are undefined as if they were never initialized.

2.2.6 Interaction Point Binding.

The connect-statement is used to specify the association or binding of interaction points, that is, to bind 1) an internal interaction point of a parent to an external interaction point of one of its children, 2) the external interaction points of two of children, or 3) two internal interaction points of a module. The actions performed by the a connect- statement are the binding of the pair of interaction points specified, as well as the binding of queues to the endpoints of both the interaction points specified. The effect of the binding operation is that the names of the interaction points become synonyms for queues within the modules where the interaction points are specified.

The disconnect-statement is used to unbind interaction points bound by an connect-statement. After a disconnect-statement is executed, any interactions queued for reception remain in the queue of the receiving module. They are only destroyed if the module itself is released.

The attach- and detach-statements are other statements used to bind and unbind interaction points. They differ from the connect- and disconnect-statement in the condition of the queue before and after their use, and that attach- and detach-statements only reference external interaction points of module instances. An attach-statement unbinds the queue from the endpoint of the first specified interaction point, binds the pair of interaction points, and binds the queue to the second specified interaction point. In common with the connect-statement, the names of the interaction points become synonyms for the queues with which they are associated.

The detach-statement unbinds interaction points bound by the attach-statement by performing the following actions: the pair of interaction points

are unbound, the queue from the endpoint of the interaction point specified is unbound, the queue is bound to the endpoint of the interaction point of the module executing the statement, and any interaction points at a lower level in the module hierarchy remain bound to the interaction points of the child but there are no queues at the endpoints to receive interactions.

Interaction point binding is demonstrated by an example (modified from the draft international standard [ISO87]).

The nested modules "*A*", "*B*", "*C*" and "*D*" have interaction points "*a*", "*b*", "*c*" and "*d*", respectively. While interaction point "*a*" is internal to the module "*A*", the interaction points "*b*", "*c*" and "*d*" are external to their respective modules. The sequence of binding operations

connect *a* **to** *B.b*; (* From within A *)
attach *b* **to** *C.c*; (* From within B *)
attach *c* **to** *D.d*; (* From within C *)

links the interaction points "*a*", "*b*" and "*c*" to interaction point "*d*" (See Figure 2.3.). "*a*" and "*d*" are connection endpoints, which means that any interactions previously queued at "*b*" or "*c*" are now queued at "*d*" and are no longer visible at "*b*" or "*c*".

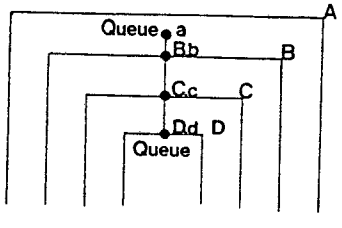

Figure 2.3: Graphic Representation of the Interaction Point Binding.

After module "*B*" executes the operation
detach *b*; (* Or the equivalent detach C.c; *)
the interaction points associated with the detach operation, namely "*b*" and "*c*", are unbound and the interactions in the queue at interaction point "*d*" which have passed through interaction point "*b*", but not through "*c*", are placed in a queue in module "*B*" for interaction point "*b*" (See Figure 2.4.).

Interactions from module "*A*" through interaction point "*b*" are now received by module "*B*" and placed in its queue.

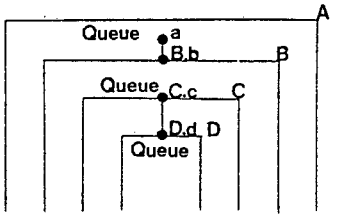

Figure 2.4: Interaction Point Binding after a Detach Operation.

2.2.7 Quantifiers.

Because it is possible for more than one instance of a module to exist during execution, Estelle has quantifiers to make dynamic evaluation of the instances possible. The all-statement is a repetitive statement which allows all instances of a module to be accessed. The exist- statement makes it possible to determine whether an instance of an object satisfying criteria exists. The forone-statement searches for an instance of an object given certain criteria.

2.2.8 Implementation Defined Elements.

Another facility of Estelle is the ability to express implementation defined constants, types and procedural elements. Constants can be defined as: "**const** X = **any** *integer*", which indicates that the value of the constant can be defined by the implementor. Likewise, the type definition: "**type** Y = ...", indicates that the type is left to the implementor. For the case of procedures and functions the directive "primitive" indicates that the functionality of the procedure or function is left to the implementor. For example, a type "*DataType* = ...;" used in a protocol data unit is not needed to define a protocol. The implementor can specify the format of the *DataType* type.

Appendix A contains an annoted grammar of Estelle, while Appendix B notes the changes made to standard Pascal to arrive at the subset used.

2.3 Criticism of Estelle.

Before discussing criticism of specific features of Estelle, it is fruitful to consider criticism based on the specification/implementation language duality.

2.3.1 The Duality of Estelle.

The purpose of Estelle is to provide a Formal Description Technique for the specification of communication protocols and services. Estelle is addressed at two groups of people, namely protocol designers and implementors [ISO87]. That is, Estelle is designed to be not only a specification language, but also an implementation language which can be processed automatically.

While protocols defined within the rigid structure of the semantically well-described constructs will be specified in an unambiguous way, it can be argued that, because Estelle is modeled so closely to specific implementation techniques and in particular the use of finite state machines, there is a loss of generality for the specifier, who must contain a specification to an implementation-orientated approach. It can also be argued that a programming language-like language is not sufficient for the specification of systems with temporal properties.

On the other hand, the use of Estelle as an implementation language for communication protocols and services is effected by its other role as a specification language in that relatively small protocols need lengthy specifications.

The duality of Estelle therefore results in it having properties which make it clumsy for implementation and also constraining for specification.

2.3.2 Specific Criticisms.

While the discarding of some constructs from the Pascal subset underlying Estelle is welcomed, a number of other constructs left in could also have been removed.

The use of reals in communication protocols and services is surely so uncommon as to make their inclusion a doubtful decision. Their inclusion results in substantial addition overhead during compilation.

The inclusion of subranges in a strongly typed language can be argued against on the basis that types can be introduced that are either disjoint, contained in other subranges, or even overlapping [Bri82]. Run-time testing is necessary to decide whether two types are compatible.

Similar arguments are possible for sets which have members defined by subranges as compatibility can not be tested statically, and only with great difficulty during run-time execution.

In the design of Edison [Bri82], Brinch Hansen noted that explicit dynamic retyping seemed preferable to the use of variant records, while also questioning the use of pointers. In common with the tagfields of variant

records, pointers need to be initialized to nil automatically to allow the detection of run-time errors.

Type synonyms are also a debatable inclusion in Estelle as the compatibility rules define that the type definition "type $T_1 = T_2$;" implies that two variables of type T_1 and T_2 respectively are incompatible. For the case where T_2 is the standard type *integer*, this results in two sets of incompatible integer values.

The inclusion of the goto-statement in the language can also be considered debatable, given its notorious reputation.

One can criticize the inclusion of the for-statement, with-statement and case-statement, as these statements represent constructs which can be expressed in terms of other statements. While their inclusion contributes to the readability of specifications, it also contributes to the size of the language, and consequently, to the size of the language processor.

From the viewpoint of the implementor of an Estelle compiler, the form of forone-, existone- and all-statements is difficult, as these statements can contain either a domain-list or a module-domain, necessitating at least three symbol look-ahead in some cases to distinguish between the two forms. As the domain-list form of the statements can be expressed in conventional constructs their inclusion contributes nothing to the language and makes compilation messy.

In addition, the case of assignment-statements semantic information is necessary to distinguish between syntactic constructs. While an expression, say "$V_1 := V_2 + V_3$" is syntactically correct if V_1 is a character, and V_2 and V_3 are integers, it is semantically incorrect. However, in the case where V_1, V_2 and V_3 are module variables, the assignment is syntactically incorrect as an expression is not permissible on the right-hand side of a module variable assignment. Only semantic information can be used for correctly checking syntax, forcing the compiler to collect semantic information even when the "**any** *constant*" or "..." constructs have made it impossible to perform complete semantic analysis of the specification.

A number of additional problems with Estelle become obvious once an implementation is attempted. These problems will be discussed in conjunction with the aspects of the compilation that they make difficult.

2.4 Advantages of Estelle.

The close association between the finite state machine model underlying Estelle and the language itself is of great value, especially as FSM's are a common method of protocol description, making it relatively easy to understand for protocol designers and implementors.

One of the most valuable features of the Estelle language is that the formal semantics have already been defined. This provides for a consistent language definition and provides a sound basis upon which analysis, testing and performance evaluation can be performed.

Another advantage is the support that is given for various levels of abstraction. This means that the various levels of detail can be expressed in Estelle, from high level descriptions with little detail for specification to low level descriptions with abundant detail needed for implementation. The ability to specify interconnection of modules, the interactions between modules, and the effect of module creation and termination on interactions in queues is also valuable.

Estelle was also designed to have features contributing to the verification of protocols specified with the FDT. An example is the fact that the clauses guarding transitions have no side effects.

While advantages and disadvantages of the FDT Estelle have been discussed, no opinion concerning the value of Estelle has been expressed. This omission will be corrected in chapter 10.

The following chapter discusses general issues concerning compilers.

Chapter 3

Compilers.

The principle source of information for a compiler implementor is undoubtly the "New Dragon Book", Aho, Sethi and Ullman's "Compilers – Principles, Techniques and Tools" [Aho86]. For the Estelle compiler this was also the case. Brinch Hansen's "Brinch Hansen on Pascal Compilers" [Bri85] was used extensively as the source of a practical compiler. Repeated references to these two sources are therefore unavoidable.

A compiler is a program, written in some programming language, which accepts text in one language, the source language and produces text in another language, the target language.

The source language is usually a human-readable language, while the target language is a machine-executable language. The translation process usually has two phases namely an analysis phase, which will be discussed next, and a synthesis phase.

3.1 Source Text Analysis.

During the analysis phase the source language text is analyzed to discover its semantics. The syntax is also analyzed, but as an aid to discovering the semantics. There are thus three interrelated forms of analysis, namely **lexical**, **syntactic** and **semantic**. These analysis phases are also known as **linear**, **hierarchical** and **semantic** analysis [Aho86].

3.1.1 Lexical Analysis.

Lexical analysis concerns the process of associating strings of characters with symbols of an input alphabet, or **tokens**. Extra characters such as blanks,

and comments are also removed from the token stream which is sent to the rest of the compiler. DeRemer [DeR74] calls the former **scanning** and the latter process **screening**. However, scanning is usually used as a synonym for lexical analysis.

Although the implementation of lexical analysis is a trivial exercise, analyzers are sometimes automatically generated from lexical grammars. For example, LEX, a lexical analyzer generator is available on the Unix operating system and is used in conjunction with a parser generator, YACC [Aho86]. To hand-code lexical analyzers, a Finite State Diagram which corresponds to a lexical grammar (See [DeR74]) can be used to guide the implementor [Gri71]. The hand-written analyzer usually takes the form of a case-statement within a loop.

3.1.2 Syntactic Analysis.

During syntactic analysis, or **parsing**, tokens are grouped hierarchically into nested collections with collective meaning [Aho86]. Thus, it concerns the identification of syntactic structures in the source text.

Parsing, and its association with formal language theory, is discussed extensively in the literature concerning compilers ([Gri71], [Aho72], [Hop79], [Tre85], [Aho86]). For the purposes of this thesis only the characteristics of various methods are discussed.

There are basically two approaches to parsing, namely top- down and bottom-up parsing. (There are methods which can not be classified into either group.) While top-down parsing concerns an attempt to construct a syntax tree by starting at the root (start symbol) and proceeding down towards the leaves (symbols) of a sentence, bottom-up parsing concerns completion of the syntax tree by starting at the leaves and attempting to reduce to get the root.

Bottom-up parsing algorithms in general can recognize a larger class of languages, but are difficult to understand and make code generation difficult. Bottom-up tabular methods need large tables to perform their task.

Top-down parsing algorithms are more natural, (easily understood) and code generation is easier to perform. In tabular top-down methods the tables are much smaller that those for the bottom-up methods. Unfortunately, top-down tabular methods can only recognize a special class of languages.

Although a large number of methods exist in both classes, three methods will be discussed, namely: **recursive-descent** (top-down), **LL(1) parsing** (top-down) and **LR(1) parsing** (bottom-up).

Recursive-descent parsing ([Gri71], [Wir71], [Aho72], [McK74], [Bri85], [Tre85], [Aho86]) is an important method to consider, not only because of its popularity, but also its association with Pascal which plays such an important role in Estelle. Recursive-descent parsing is a widely used method with examples such as the original Pascal compiler [Wir71], the P4 Pascal compiler (thoroughly discussed in [Pem82], [Pem82a]), the Edison compiler [Bri82] and the Pascal- compiler [Bri85]. Each procedure of a recursive- descent parser recognizes a single language structure and the procedures call each other recursively to recognize structures. In contrast to the other methods to be discussed, a recursive-descent parser has an implicit stack provided by the implementation language's activation records. This means that no stack mechanism has to be programmed, simplifying not only information storage for syntactic analysis, but also for semantic analysis. A special case of recursive-descent parsing in which no backtracking is required is known as **predictive parsing** [Aho86].

LL(1) parsing ([Gri71], [Aho72], [Gri74], [Tre85], [Aho86]) concerns the parsing of language defined by a LL(1) grammar, a special case of LL(k) grammars. The two "L"s in "LL(1)" stand for scanning from left to right, and for producing a leftmost derivation, respectively, while the "1" stands for a lookahead of one symbol to make decisions concerning which parser action to take. The method is closely associated with the recursive-descent parsing method as it is also known as **nonrecursive predictive parsing** [Aho86]. In common with LR(1) parsers, LL(1) parsers simulate theoretical machines of language theory called push-down automata. The LL(1) parsers implemented in this fashion, are simple and easy to understand. A disadvantage of the LL(1) parsing method is that due to left-recursion it is often awkward or difficult to modify a grammar to satisfy the requirements of an LL(1) grammar [Tre85].

LR(1) parsing ([Gri71], [Aho72], [Hor74], [Aho74], [Tre85], [Aho86]) is a special case of LR(k) parsing. The "LR(1)" stands for left to right scan, rightmost derivation, and lookahead of one symbol. Three modifications of LR parsing are important, namely **SLR** (simple LR), **Canonical LR**, and **LALR** (lookahead LR). LALR parsing is a compromise between the other two in terms of power and cost, being more more difficult to implement, but more powerful than SLR, and less powerful than canonical LR, but also less expensive. LALR is sufficiently powerful to implement most programming languages, with some effort, can be implemented efficiently [Aho86]. An advantage of LR parsing over LL(1) and recursive-descent parsing is that class of grammars that can be parsed using LR methods is a proper superset of

the class of grammars that can be parsed with predictive parsers. A problem associated with LR parsers is that it requires to much effort to construct a parser for a typical programming language by hand. Parser generators are therefore necessary. Yacc, "yet another compiler-compiler", a LALR parser generator on the popular operating system UNIX is an extensively used example.

A problem associated with tabular parsing methods is the size of the parsing tables when the language is substantial. Because the tables are sparse, a large amount of memory is wasted. Various methods ([Aho74], [Hor74], [Den84], [Aho86], [Al–86]) are used to transform the tables resulting in a substantial saving of memory. A compaction method based on [Den84] was used in the LL(1) parsing system developed for the prototype Estelle compiler [vDi87].

The debugging of grammars used in parsers is considered to be easier for LL(1) and recursive-descent parsers than for LR parsers, because of the complicated process of creating LR parser tables [Tre85]. There is very little to choose between error detection for LL(1) and LR parsing, while recursive-descent is hand implemented, and error detection depends on the implementor's accuracy. There is actually very little to choose between the methods concerning storage requirements and speed, although Waite and Carter have demonstrated that recursive-descent will always be superior to generated parsers if the parser table interpreters are implemented in a high-level language [Wai85].

Other factors which are important in parsers are the availability of parser generators and the level of error detection, reporting and recovery needed [Tre85]. The use of Lex and Yacc on UNIX should be considered and a LL(1) parser generator can be implemented with small effort if no parser generator is available. A straightforward way of getting the job done is to use a recursive-descent parser [Tre85].

3.1.3 Semantic Analysis.

Semantic analysis associates the syntactic structure found in syntactic analysis with "meaning". During this phase checks are made to enforce the rules governing the meaning that the source text may have. Semantic analysis is chiefly concerned with the analysis of two types of rules, namely **scope** and **type** rules. Other checks are made for **control flow**, **uniqueness**, and **name-related** checks. For example, a control flow check needed in Estelle is a check for the existence of a label for a goto statement. Uniqueness

checks needed in Estelle are, for example, that labels in a case statement are distinct, that variables may only be declared once, that the identifiers in an enumeration type are unique, and that, in the clause part of a transition, clauses are not repeated. An example of a name-related check in Estelle is for the interaction argument list of a when clause, where, if the optional interaction argument list is given, the arguments must all be given, and be in the same order as those in the interaction definition.

3.1.3.1. Scope Analysis.

In many languages it is necessary to declare a name before use. The declaration of the name only holds for a portion of the source text known as the **scope** of the name. In its scope all uses of a name refer to the same object [Hor84].

Symbol tables are used to analyze the scope of objects in a specification. When a declaration is encountered, an entry is made for the name. When the region associated with the declaration ends, the entry is removed from the symbol table. Every occurrence of a name is looked up in the symbol table and therefore an occurrence of a name outside the scope of its declaration will be found to be erroneous. In section 3.3. symbol tables will be discussed.

3.1.3.2. Type Analysis.

Type analysis is concerned with the checking of the rules of the type system of the source language. A type system defines the standard data types and the facilities for defining new data types from previously defined types and standard types [Hor84]. A data type consists of objects and operations which are permissible upon the objects. For example, amongst others, the operations addition, subtraction, multiplication and division are defined between two variables of the data type integer, which occurs in many languages. When an addition is performed between two integers, the type checker will validate the operation as being a legal operation with an integer result. However, if a logical "and" operation is attempted between two integers, the type checker will signal a breach of the type system rules. In some cases, rules of the type system explicitly allow different types to used in operations together. For example, reals and integers can added together in an expression by **coercion**, resulting in a real result. Type definitions, which allow the definition of new data types from existing types, are also checked during type analysis.

The checks mentioned above are all **static** type checks, which can be checked during compile time. Some checks, called **dynamic** checks, can only be performed during execution of the object code. For example, the fields in a variant part of a variant record can only be accessed if the tag-field associated with the variant part has the value corresponding with the existence of the variant part. Thus, before accessing a field in a variant part, the tag must be checked, otherwise the field has no legal type.

Symbol tables are used for type analysis in much the same way as they are used for scope analysis. A definition of a constant, type or variable results in an entry with information concerning the type of the object being made in the symbol table. Every occurrence of the object is then checked to see if the type rules are upheld.

3.1.4 Error Handling.

During the analysis of the source text errors are often encountered: **lexical errors** such as illegal characters, identifiers which are too long and numerical values which are too big to be represented; **syntactic errors** where the source text does not conform to the syntax of the source language; and **semantic errors** where type or scope rules have been violated.

Different levels of error handling can be defined, namely **no response**, **first error**, **error recovery**, **error repair** and **error correction** [Tre85]. In the first level the compiler does not detect or report the error – a totally unacceptable state of affairs. In the second level only the first error is detected and reported. Although this is considered to be an unacceptable response [Tre85], it is common in many small compilers. For example, the Turbo family of compilers. The next level is error recovery, where an error is detected, reported and the compiler recovers to a state in which additional errors can be reported. A problem that can occur with this type of error handling is the generation of spurious error messages. It is therefore necessary to prevent the loss of information caused by an error, and to ensure that further errors propagated by an error are avoided. The error repair method of error handling consists of the modification of the present input to make it syntactically valid. Thus, errors are detected, reported, the input is modified, and valid object code is generated. This level is also known as **error correction**, although this term should be reserved for true error correction, namely a method which takes an erroneous program and produces a correct object program. This last method is beyond present techniques.

Error detection during lexical analysis is fairly simple, but in the case of

a misspelt identifier, keyword, or operator is extremely difficult, as syntactic and semantic information is necessary to identify the error. This type of error is usually detected during syntactic or semantic analysis and erroneously reported to be a syntactic or semantic error. However, some compilers exist which detect these errors, report them, and, by deleting an extraneous character, inserting a missing character, replacing an incorrect character by a correct one, or transposing two adjacent characters, repair them [Aho86]. Spenke et al. [Spe84] described an LL(1) parser generator which included this type of error repair. In general, however, it is sufficient to report and recovery from the syntactic or semantic errors caused by the misspellings.

In many compilers error detection is centered around syntactic analysis as many errors are syntactic in nature, but also because of the precision of modern parsing methods [Aho86]. In addition, as has been mentioned above, lexical errors which cannot be detected easily are detected as syntactic errors. As the detection and recovery of syntactic errors differs between parsing methods, error handling for the three parsing methods discussed in section 3.1.2. will be examined.

LL(1) parsers, in common with LR parsers, have the **valid prefix property**, which implies that if the parser does not detect an error in the first portion X of a program XY, then there must exist some string of symbols W such that XW is a valid program [Tre85]. This has distinct advantages, as the sooner a compiler can detect an error, the better the chances are that the error can be correctly reported and adequate actions taken to recover. LL(1) parsers detect errors by examining the parser table. When a nonterminal- terminal pair, say $M(A, a_k)$, is not represented in the table, an error has occurred. By the valid prefix property, because no error occurred for the first portion $a_1a_2...a_{k-1}$ of an input string $a_1a_2...a_n$, there must be some string $a_ka_{k+1}...a_m$ such that $a_1a_2...a_m$ is a valid string of the language [Tre85]. Therefore, by modifying the unparsed input string $a_ka_{k+1}...a_n$, the parser can recover from errors.

The method of modification is important, as it effects the performance of the parser and the recovery method. Two approaches can be identified, namely an **ad hoc** approach and a **systematic** approach. With an ad hoc approach error entries are added to the parser table, with each entry formulated by the compiler writer. This method, known as **phase level recovery** as recovery is performed by making local correction to the remaining input at phase level, has the advantage that tailor-made error messages can be provided for each entry, hopefully providing a clear picture of the error. In addition, it makes it possible to perform modifications to the input with

heuristics suited to the type of error occurring in the compiler's environment, i.e. the type of error common to the class of user. Unfortunately, poor choice of error routines can have detrimental effects on not only the error recovery, but also the error reporting. In addition, when the language is substantial, the task of providing a large quantity of quality error routines is imposing.

With the systematic approach error reporting is provided by examining the top of the parser stack and the present input symbol. For example, if the nonterminal "<Expression>" is on the stack and the input symbol ";" is an erroneous symbol, a message of the following type can be generated:

"<Expression> expected, but ';' found".

Algorithms are used in a systematic approach to choose the modifications used for each error detected. This may involve the insertion, deletion or modification of an input symbol. The simplest systematic method of error recovery, known as **panic mode**, consists of the deletion of all input symbols until one of a set of synchronizing symbols is found. Naturally, the choice of the synchronization set profoundly effects the performance of the recovery method and the ability to detect further errors, as well as the generation of spurious error messages. The modifications discussed by Spenke et al. [Spe84] include use of the **immediate-error- detection property**, a stronger form of the valid prefix property, which allows errors of the form

if $a * b; -c > 0$ **then** S

to be correctly detected and reported. In addition, algorithms for **least cost repair** are discussed. Tremblay and Sorenson [Tre85] also discuss some interesting methods for error recovery with LL(1) parsers.

Because LR parsers also has the valid prefix property, error recovery similar to that for LL(1) parsers can be used, i.e. the modification of the input, or the alteration of the parser state. Error routines can also be used in the parser table. Tremblay and Sorenson [Tre85] and Aho et al. [Aho86] discuss some methods in detail.

A standard method for error detection and recovery for recursive-descent parsers has evolved [Tre85]. The method, which is discussed in detail by Wirth [Wir76], consists of modifications being made to the recursive procedures which are constructed from the grammar of the source language. Each procedure which parses a construct of the language insists on leaving the current input symbol in a well- defined syntactic class, the **stop set**. The stop set is the union of all the sets of first symbols of constructs which can follow the present construct, together with a set of symbols which always terminate a construct. For example, one symbol which always is in

the stop set is the end of text symbol, a token denoting the source text's end. Errors are detected when the current input symbol is not in the stop set, indicating that no legal construct follows. By considering single symbols to be constructs, errors are detected inside constructs. Note that this is an implementation of a form of panic mode recovery. This method has been applied to a number of Pascal compilers and Pascal- like language's compilers, for example, the original Pascal compiler [Wir71], the P4 Pascal compiler [Pem82], [Pem82a], the Edison compiler [Bri82] and the Pascal-compiler [Bri85].

A method of recovery which has not been discussed is the use of error productions. By adding error productions to the grammar, common errors can be parsed as if correct, and an error message reported [Aho86].

The detection of semantic errors in the source text of a program can not always be detected during compilation. In some cases, for example, in the case of an array's subscripts, semantic errors can only be detected by the run- time environment. However, semantic errors detected during compilation must be handled in much the same way as syntactic analysis, i.e. reporting and recovery. Because semantic checks are performed to check specific semantic rules, adequate information is available to report the precise nature of the error. Recovery, i.e. the avoidance of error propagation, can be performed by declaring undeclared objects or incompatible objects with a universal attribute. This prevents further errors caused by a undefined object or type incompatibilities.

Horning [Hor74a] discusses the role messages play in a compiler - not only error messages, but also informative messages indicating the lack of errors or warnings. Error messages are designed to provide the user with as much information concerning errors as possible. The location in the source where an error was detected should be reported, along with the suspected nature of the error [Aho86]. It is often the case that errors which are detected in close proximity have the same cause, and some of the error messages are spurious. Usually, such error messages are suppressed. Note that the extent of the information which can be provided by the compiler is governed by the parsing method. Every compilation should provide information such as the source language, target machine, the compiler name and version, date when the compiler was created, time of current compilation, options in effect and options suppressed [Hor74a]. In addition, optional listings, cross-referencing and symbol table dumps should be considered.

Other errors which have not been discussed yet are internal errors, i.e. the failure of the compiler itself. While the errors discussed before all oc-

curred in the analysis phase of the compiler, these errors occur in any phase. Typically, these errors are tables which are not large enough to allow a large program to be compiled, stack overflows, insufficient heap memory, and unfortunately, errors in the compiler, i.e. "bugs". While the former errors can be considered legitimate or permissible if they occur only when

the compiler is used for purposes for which it is not designed, the latter errors are inexcusable.

The synthesis phase of the compiler, also known as the **back end**, is discussed in the next section. (The analysis phase is also known as the **front end**.)

3.2 Target Text Synthesis.

During the synthesis phase text is synthesized to produce text semantically equivalent to the source text in the target language. This phase, commonly known as **code generation**, uses the syntactic structure and semantic information identified in the synthesis phase to produce code. Another phase in the synthesis of the object text is **code optimization**, where transformations are made on the produced code to make it run faster or with less memory.

3.2.1 Code Generation.

The generation of object code from the syntactic and semantic information provided by the front end is usually associated with the generation of an intermediate, machine independent language. This intermediate language can take a number of forms, amongst others **Polish notation**, **n-tuples**, **abstract syntax trees**, **threaded code**, and **pseudo or abstract machine code** [Tre85]. When intermediate code is produced, another phase is needed to produce absolute code. Some of the issues involved in this last phase are **memory management**, **instruction selection**, **register allocation**, and **evaluation order** [Aho86].

Memory management is concerned with associating a run time address with a data object, where use is made of the symbol table to retrieve information such as a relative address of the object and the amount of memory needed to represent the object. During code generation the compiler must generate code using addresses defined by some memory allocation scheme. In section 3.4. memory management schemes will be discussed.

Instruction selection is concerned with the generation of the optimal instruction or sequence of instructions where there is a number of possibilities.

Operations making use of registers of the target machine are faster than those using memory operands. Register allocation therefore concerns the utilization of registers in the generated code. The problem can be divided into two subproblems, namely **register allocation** and **register assignment** [Aho86]. Register allocation is a phase of the problem where variables are selected as those that can be advantageously associated with registers, while register assignment concerns the actual association of variables with specific registers. While it is already difficult to find an optimal solution as the problem is NP-complete [Aho86], selection is often hampered by register usage conventions forced by the hardware or operating system. For example, on the IBM-PC/AT, for which the Estelle Compiler generates code, only certain registers can be used as index registers and some operations require the operands to be in certain registers while the result is placed in another. This obviously makes register allocation difficult.

By choosing the order of evaluation of computations, the efficiency of the target code can be effected. Picking a best evaluation order is also an NP-complete problem [Aho86].

In addition, it is necessary to take the form of the input, as well as the output of the final code generator into account. The output can take three forms namely absolute or relocatable machine language and assembly language [Aho86]. Absolute machine language is common for compilers for students of programming, as absolute machine can be placed at a fixed memory address and executed immediately. This is very valuable when small programs need to be repeatedly compiled during debugging. Relocatable machine language has the advantage of allowing separate compilation of subprograms. However, relocatable machine language programs require linking and loading before execution. The generation of assembly language is very simple, but has the disadvantage that assembly is needed before execution.

Data structures and algorithms for the implementation of code generation can be found in the literature ([Wai74a], [Amm77], [Tre85], [Aho86]).

3.2.2 Code Optimization.

The transformations made on the generated code can never guarantee optimal code, and the term "optimization" is therefore a misnomer. "Optimization" should really be called "improvement".

There are two types of code-improving transformations, called **machine-dependent** and **machine-independent optimization** respectively. The former are transformations that are made to take advantage of a specific machine's features. For example, special purpose instructions such as an instruction to increment the contents of a register or a storage location should be used where possible. In addition, a register can be used as a storage location where a variable is frequently accessed.

Machine-independent optimizations are transformations which improve the generated code without taking the features of the target machine into consideration. The concept of a **basic block** is important for distinguishing between **local** and **global** optimizations. A basic block is a sequence of consecutive statements in which flow of control enters the at the beginning and leaves at the end without halt or possibility of branching except at the end [Aho86]. Local optimizations are those optimizations which can be applied by examining instructions within a basic block, while other optimizations are called global.

Examples of local optimizations are those which can be classified as function preserving transformations such as **common subexpression elimination, copy propagation, dead- code elimination, constant folding** [Aho86] and **constant propagation** [Tre85], and loop optimizations such as **code motion, induction-variable elimination, unswitching, strength reduction** [Aho86], and **loop unrolling** [Tre85].

Common subexpressions (also known as **redundant subexpressions** [Tre85]) are subexpressions which occur more than once in a program and need only be evaluated once. For example, in the expression

$$(A + B) * (C + D) + (A + B) * E$$

the addition "$A + B$" need only be performed once. It can be argued that programmers are taught to program in a non- redundant manner, thus questioning the value of this transformation. However, array addressing often provides redundant subexpressions over which the programmer has no control, and programming redundantly often improves readability, a common requirement of a program. For example,

$$A[i,j] := A[i,j] + B[i,j]$$

is an instance of array addressing where the address "$A[i,j]$" need only be evaluated once, and if the array variable "A" and "B" have the same dimensions, the subexpression for the offset need only be evaluated once.

Copy propagation is concerned with the use of "t" wherever possible for every occurrence of "A" after a statement "$A := B + C$", where "t" is the result of the expression "$B + C$". This transformation often causes dead-code to be formed, which can then be removed. **Variable folding** is used in the same way where the assignment "$A := B$" occurs for "B", a variable.

Dead code is code that has no purpose. For example, a statement which conditionally prints debug information if a boolean variable is true, can be removed if the variable is obviously false at that stage. Say the statements

debug := *false*;
if *debug* **then** S;

occur in a program. Then from copy propagation the second statement would become

if *false* **then** S;

which can be removed as "S" will never be executed.

Constant folding is used where expressions can be replaced by their values at compile time. For example, the assignment

$A := 1 + 2 + 3;$

can be replaced by the assignment

$A := 6;$

Subexpressions can also be reduced via constant folding. For example, the expression

$1 + 2 + A + 3 + 4 + B + 5 + 6$

can be reduced to the expression

$3 + A + 7 + B + 11$

Constant propagation is similar to copy propagation as the values resulting from constant folding are replaced in all possible occurrences of the variable in an assignment with a value produced by constant folding.

Code motion (also known as **frequency reduction**) is used in loop optimization where code known as the **invariant** is removed from the loop and performed before the loop. Thus, in cases where the code inside the loop is executed more than once the code is vastly improved, for the case

where the code is executed once there is no difference, and for the case where the loop code is not executed, the performance is degraded.

Induction-variable elimination, or **loop fusion**, is used where two or more loops can be combined into a single loop, thereby reducing the number of test and increment/decrement instructions executed. Induction variables are variables which are associated in such a way that when one increases/decreases by a certain amount, the other increases/decreases by an amount proportional to the other's increase or decrease.

Unswitching is used when it is possible to divide a loop into two loops with guards which are mutually exclusive. For example, the loop

for i := 1 **to** 10 **do**
  if i <= 5 **then** S_1
  else S_2;

can be replaced by the loops

for i := 1 **to** 5 **do** S_1;
for i := 6 **to** 10 **do** S_2;

Strength reduction concerns the replacement of a computationally expensive operator with a less expensive computation. For example, multiplication is often replaced by addition.

Loop unrolling is used where it is advantageous to repeat the code inside a loop. For example, the loop

for i := 1 **to** 3 **do** $a[i] := 1$;

can be replaced by the three statements

$a[1] := 1$;
$a[2] := 1$;
$a[3] := 1$;

which saves the incrementation and testing code for the loop execution. However, in return for the execution speed gained, more space is required.

Examples of global optimizations are common subexpression elimination, constant folding, constant propagation and variable folding, which are also used for local optimizations, as well as redundant statement elimination.

Redundant statement elimination is used when entire statements can be removed. For example, the assignments

$A := B;$
$A := C;$

result in the first assignment being redundant.

Other optimizations which can be used without reference to basic blocks are **peephole optimization, array linearization,** the **rearranging** of expressions, **procedure call optimization,** and **procedure integration.**

Peephole optimization is a method used to improve code by examining small sequences or "peepholes" of code [Aho86]. Typically, the transformations performed during peephole optimization are **redundant-instruction elimination, flow-of- control optimizations, algebraic simplifications** and **use of special machine instructions.** An example of redundant- instruction elimination is for the sequence

Load $Register_0$, A
Store $Register_0$, A

where the second instruction is redundant as the values in A and $Register_0$ are the same after the first instruction.

Preliminary code generation often produces redundant flow- of-control instructions such as the following sequence

Goto L_1
L_1: instruction

can be changed to the single instruction

L_1: instruction

Algebraic simplification can be used when, for example, an identity element is used in an operation. For example, the assignment

$A := B + 0;$

can be simplified to

$A := B;$

When multi-dimensional arrays are accessed, array linearization can be used to reduce the number of multiplications.

The order of evaluation of expressions can be rearranged to improve the speed of execution or the amount of temporary storage used.

Procedure and functions calls can often be improved by dividing the parameters into different categories and having special purpose procedures for each class of parameters.

By treating procedure and function definitions as macro definitions, i.e. replacing the procedure or function call with the body thereof, the poor implementation of procedure calls can be improved by removing the overhead involved.

Code optimization has been discussed in this section and the data structures and algorithm needed to perform the transformations can be found in the literature. ([Wai74b], [Tre85], [Aho86]).The design, contents and use of symbol tables is discussed in the next section.

3.3 Symbol Tables.

The design of a symbol table is an important consideration in the implementation of a compiler as the information in the symbol table is updated and accessed during type analysis, scope analysis, code generation and in some cases during syntactic analysis.

Typically, the information stored in the symbol table include some of the following attributes: **name, run time address, type, dimension** or **number of procedure parameters, source line number** where object was declared, source line numbers at which the object was referenced, and a **link field** for listing in alphabetical order [Tre85]. Naturally, for different types of objects different attributes must be stored. Often most of the attributes are stored outside the table, allowing uniform symbol table entries. Another solution is the use of variant records, or, when dynamic retyping is available, different templates which allow different attributes to be stored in an uniform entry.

Three operations are performed on symbol tables, namely the **insertion** of a new entry, **searching** for an existing entry or the **deletion** of an existing entry. Deletions and searches apply to the most recent entry with the search key, to provide for the closest-containing block scope rule. Because a large proportion of compile time is spent on interrogation of the symbol table, the symbol table design is very important.

There are various approaches to symbol table construction, notably those based on **hashing** and **tree** or **list structures**.

Symbol tables based on linked lists or trees have the advantage of flexibility. An additional advantage is that storage is dynamically allocated, and

only the required storage is used, an important consideration when available memory is limited. Symbol tables based on linked lists are the simplest and easiest to implement, but unfortunately, have extremely poor performance characteristics when there is a large number of entries as the time is proportional to n, the number of entries [Aho83].

Insertions into linked list constructed symbol tables are implemented by linking the new entry onto the start of the list. Table lookup is achieved by examining entries starting with the most recent entry, i.e. the entry at the root of the linked list, thereby making it possible to find the most recent instance of a variable.

Binary trees are the most common tree structure used for symbol tables. For a binary tree symbol table with entries in alphabetical order, insertions are performed by starting at the root and going left if the new entry is greater than the present entry and right if it is less, until a move to a non-existent entry is attempted, where the entry is added. Lookup is achieved by adding a test for the relevant entry.

Note that the performance of trees can degenerated to that of linked lists, or worse, if the order of entries is adverse. For example, if constants, types, variables, procedures and functions are declared with names in alphabetical order, the tree has one long right branch, and can be considered a linked list. To prevent this sad state of affairs, tree balancing algorithms ([Knu73], [Aho83], [Tre85]), are used to rearrange trees after each insertion to ensure that the average length of the branches of the tree are within certain limits, thus ensuring that the time for insertions and lookup is proportional to $O(logn)$ [Aho83].

Tree structured symbol tables can not be used for block structured languages unless some method is used to preserve the order of definition of entries. Usually, the method used is to have a stack of root pointers, each being the root of a tree for a program block.

Hashing is the process of applying a hash function h to a key K to calculate an address in a hash table where the information corresponding to K can be found. Where the information is static it fairly simple to arrive at a function h so that $h(K)$ is unique for each possible K. This implies an extremely fast access of entries. Unfortunately, the "birthday paradox", (See [Knu73]), indicates that the probability is good that for some distinct keys $K_i <> K_j$ there is a collision as $h(K_i) = h(K_j)$. Therefore, for a hash based symbol table, a hash function h must be chosen, as well as a collision resolution method. There are basically two methods of hashing with collision resolution, **open hashing** and **closed hashing**.

With open hashing the hash table, also known as a **bucket table**, contains n **buckets**, where n is the number of possible values for $h(K)$ for all K, and each bucket contains the root of a linked list of entries which all have the same hashed value. Collisions are therefore solved by adding the entry into the linked list associated with the hash value. When searching for an entry, the key is hashed and then the associated linked list is searched. Note that this method, also known as **direct chaining**, degenerates when the collisions are frequent. A large bucket table can contribute to ensuring that the list are short, but the choice of hashing function is also important.

In closed hashing the buckets contain all the entries and when collisions occur, another hash is applied until an empty bucket is found, where the entry is added. When all possible unique buckets for the key have been attempted, the table is full. Searching is performed by hashing and rehashing in the same way, until the entry is found, an empty bucket is encountered, or all buckets have been exhausted. One example of the method is known as **linear probing**, where $H(K)+i \bmod m, i = 1, 2, \ldots n$ is used as a rehash, until $h(K)+i = h(K)$.

A simple and commonly used hashing function for symbol tables is to first add the ascii values of the characters of the key, i.e. an identifier, and then take the remainder of a division by m, where m is prime and the size of the hash table.

Knuth [Knu73] discusses the merits of various hashing methods, and a number of hashing functions. Analysis of the various hashing methods provides an indication of their performance, and a method can therefore be chosen to satisfy the requirements of the symbol table. In addition, by choosing m, the size of the hash table, the average number of comparisons for a certain **loading factor**, (the ratio of m to n the number of entries), can be chosen (See [Knu73], [Bri85]).

The use of hashing for a symbol table for a block structured language may seem to be problematic as hashing is not order- preserving. However, by using a stack in much the same way as for tree structured symbol tables, a scope link which chains all the entries for a block can be created, solving the problem [Aho86].

The run time environments associated with compiler systems are discussed in the next section.

3.4 The Run Time Environment.

The run time environment of object code generated by a compiler is the software providing functions necessary for the code to execute. Functions which are provided include **error detection**, **memory management** and **process management**.

Errors which can occur during run time include invalid system calls, mathematical over or underflow, division by zero, array bounds being violated, values out of subranges, invalid use of variant records and error conditions caused by memory management such as stack overflows, heap overflows and invalid pointer uses. A run time environment provides code to check these conditions and report them. In some cases code is generated by the compiler itself, with calls to a run time error handler. Information reported by the error handler can consist of the values of the machine's registers, the current instruction and the equivalent line number in the source text.

For languages without nested procedure calls **static memory management** is sufficient, i.e. the address of each object is known at compile time. Unfortunately, languages which allow nested procedure calls need a more elaborate scheme, as the precise manner in which procedure calls follow each other is not known until run time. **Dynamic storage allocation**, in which memory is allocated for an object when a block is entered, provides the solution.

Each time a block of the program is entered, an amount of memory, known as an **activation record**, is allocated from the **run time stack**. An activation record, or **AR** for short, usually contains three areas, namely the **local data area**, **parameter area** and the **display area** [Tre85]. The local data area contains all the data objects local to the block associated with the AR. The parameter area contains the parameters of a procedure or function, as well as the **return address**, which corresponds to the address in the object code where execution must commence on completion of the block, the **previous-activation-base pointer**, also known as the **dynamic link** [Bri85], which indicates the start of the AR of the block from where the current block's execution was started, and in some implementations a **return value** for a function [Tre85]. The display area of an AR is used to allow access to objects which are global to the block. It often consists of a series of pointers, each pointing to the AR of a block which is visible to the present block. In some cases it consists of a single pointer, known as the **static link** [Bri85], which points to the AR of the block which closet-

contains the current block. This second approach is much slower than the first, necessitating traveling along all the static links, to arrive at an appropriate AR [Tre85]. The return address, the dynamic link and the static link are sometimes known as the **context** of an AR [Bri85].

Code for addresses are generated by using an address relative to the beginning of an AR associated with a block. Thus, at run time an address is calculated by taking the base address of the AR in which the object is to be found and adding the object's offset. If the object is not in the current block, the display is used to access the correct AR. Because the difference in block levels is known at compile time, an address can be generated as the pair Level, Offset.

Another type of memory allocation which is often needed in programming languages is **heap memory management**. The heap is used for the allocation of memory which was not allocated at block entry. For example, in Pascal the use of "**new**(p)" implies that memory must be allocated for an object which is to be associated with the pointer p. Because large amounts of memory can be explicitly allocated in this manner, and addressing cannot be performed during compile time or at block entry, a sophisticated heap management system is needed in the run time environment. The heap manager must make provision for the allocation and retrieval of blocks of memory which are not necessarily the same size.

One solution is to always allocate the same size block for any object, or a number of them if the memory needed is bigger than the block size. Another solution is to allocate the precise amount needed, unfortunately implying that some method must be used to solve problems caused by **fragmentation** of memory, i.e. that the memory is available but not in a continuous piece of memory.

When the memory is available, a heap manager can have algorithms to choose which piece of memory to allocate, using some approach such as **first-fit** or **best-fit**. Compaction is a method in which fragments of memory are joined to produce blocks big enough to be allocated.

When memory is deallocated, some method must be applied to make **garbage blocks**, i.e. deallocated blocks, available for reallocation. Two approaches are **free-as-you-go storage release** and **garbage collection** [Tre85]. The former consists of making each block of heap storage available as soon as it is deallocated, while the second consists of collecting garbage blocks when memory has been exhausted.

A problem associated with heap storage and the reallocation of memory is the **dangling-reference problem**, namely that, although a block of

memory has been deallocated, references still remain, resulting in disaster.

Tremblay and Sorenson [Tre85], Aho et al. [Aho83] and Aho et al. [Aho86] present algorithms for the various schemes, as well as discussing the problems involved. Note that the question of memory management is also of importance in operating systems ([Bri73], [Dei84],) and similar schemes are used.

For languages which provide constructs to describe concurrent execution of processes it is necessary to provide routines in the run time environment for process management. The features of a language, for example, communication method and synchronization method, specify the routines needed to support concurrency. Some of the problems involved are **process memory allocation, process scheduling** and **interprocess communication.**

Three kinds of concurrent languages can be identified, namely **message-oriented, procedure-oriented** and **operation- oriented** languages. [And83]. Procedure-oriented languages, which are often called the **monitor model** because monitor-based languages are the most widely known form, use interprocess interaction based on shared variables. Examples of procedure- oriented languages are Concurrent Pascal [Bri75] and Edison [Bri82]. In message-oriented languages, such as CSP [Hoa85] and Estelle, processes interact via input/output or send/receive statements. Operation-oriented languages, which use the concept of a remote procedure call, interact with a combination of the previously mentioned methods. The process which sent a message synchronizes with the process which is performing a operation requested while the operation is being performed. Examples of operation-oriented languages are Ada [DoD80] and Distributed Processes [Bri78].

The requirements of an implementation of each class of concurrent programming language differ and are a worthwhile topic of discussion. However, as the design and implementation of this part of the run time environment was not in the scope of the Estelle compiler implementation, the problems will only be mentioned briefly.

The allocation of memory to processes has much the same problems as encountered for dynamic and heap memory allocation. On the one hand the size of the memory needed for all the possible processes is not always known at run time. In some languages which require all processes to be defined together, the memory can be divided up into equal size blocks. Unfortunately, when, as is the case for Estelle, processes can be created and destroyed dynamically, a more sophisticated method is needed. From dynamic memory allocation the use of relative addressing can be used, while concepts from

the heap memory manager, such as the treatment of garbage blocks, can be used for allocation and deallocation.

Scheduling involves controlling the order of execution of concurrent processes. Some of the operations performed upon processes are to **create**, **destroy**, **suspend**, or **resume** a process, **change a process's priority**, **block** a process, **wake up** a process and **dispatch** a process [Dei84]. While scheduling of processes may be defined by the programming language, it is often necessary for this to be done in the run time environment. Different types of scheduling schemes exist. For example, a scheduler may allow execution according to a **round-robin system** or by **priority** of processes. Another issue is how long a process may execute. Some systems switch processes when a system call is used, others allow a time-slice to a process before switching processes.

Information concerning a process is kept in a **process control block (PCB)**, which defines the process in the system [Dei84]. The PCB contains the **current status** of the process, a **unique identification** of the process, the **process's priority**, pointers to the **processes's allocated resources** such as memory and an area where the **registers** are saved. When processes are switched the current process's information is written to its PCB, and information read from the PCB of the process which will execute next. This process is often known as **context-switching**.

Interprocess communication is dependent on the concurrency model underlying the programming language, for example, **shared-memory**, **message passing** or **remote procedure calls**, and the run time environment provides the necessary routines to implement the model. For example, access to shared memory must be restricted and mutual exclusion of processes must be assured. With resources required by more than one process, the problem of **dead-lock** can arise, and provision must be made in the process manager to avoid, detect or resolve dead-lock. The literature concerning operating systems, for example, Deitel [Dei84] and Brinch Hansen [Bri73], is valuable for the solutions to process management problems available.

Many operating systems provide system calls for resource allocation which can be used, relieving the implementor from the above problems. However, this can be restricting as this means that, for example, scheduling of processes, is restricted to the method used by the operating system.

The next chapter provides an introduction to the Estelle compiler system.

Chapter 4

Estelle Compiler.

Different approaches to issues in compiler design have been discussed in the previous chapter and, while a particular method used in one of the phases may appear to be superior to others, the requirements and limitations of the compiler environment are the most important considerations. Before discussing the compiler design, design issues are discussed.

4.1 Design Issues.

Issues involved in the design of the Estelle compiler include the continuing changes to the Estelle language specification, the requirements of the proposed Protocol Development Workbench (PDW) [Kri87], the class of user and ease of maintenance.

Work was started on the Estelle compiler when the first draft proposal [ISO85] for Estelle was released, and has continued through the second draft proposal [ISO86] and the draft international standard [ISO87]. An attempt was made to ensure maximum flexibility in the compiler design, allowing changes to be made to bring the language compiled in line with the next language specification.

Linn et al. [Lin86], Ansart et al. [Ans87], and de Souza and Ferneda [dSo88] all discuss simulation of Estelle specifications based on their respective compilers, while de Saqui-Sanni and Courtiat [dSa87] describe an interpreter for the simulation of Estelle descriptions. The design of the PDW required that the Estelle compiler make provision for a similar system of simulation. Central to the simulation system for the Estelle compiler is the concept of a **meta- implementation**. A meta-implementation is an

implementation which is removed from a physical world, while providing a description of the protocol's environment. The environment provided may simulate, for example, errors on a transmission line [Eng86].

A meta-implementation in a simulation environment is used in automated protocol validation, testing and performance prediction. For example, an automated validation method developed by West [Wes78] was demonstrated on one layer of SNA by Scultz et al. [Sch80] using a FAPL meta- implementation [Poz82], while Engelbrecht et al. [Eng86] used a FAPL meta-implementation to demonstrate a performance prediction method proposed by Kritzinger [Kri86] on the Selective-Repeat plus Go-Back-N protocol described by Miller and Lin [Mil81]. The Estelle compiler was therefore designed to allow meta-implementations to be run, providing execution traces and statistics.

As the Estelle compiler was designed as part of the PDW, it provides for the needs of a small class of users, namely computer scientists involved in communication protocols. Features such as, for example, extensive error repair are unnecessary as the user will be very familiar with Estelle and will only need the location and nature of errors. Design goals such as correctness are, however, of much more importance considering the role the compiler plays in protocol validation, testing and performance prediction.

Ease of maintenance of the compiler was an issue in the compiler design as continuity of personnel is always a problem at a university. This implied that maintenance of the compiler would almost certainly be performed by someone other than the implementor, dictating a design policy ensuring that maintenance will be simple. Allied to this issue is the fact that changes will almost certainly be made to the compiler to provide for future validation, testing or performance predication methods. The most probable authors of changes will be computer scientists involved in communication protocols who are unfamiliar with the compiler, which is provided for in the design by following a policy of simplicity throughout.

Another design goal which received consideration was availability. As the Estelle compiler is a part of a larger system, the PDW, involving other personnel, its availability effects the schedule of other components and their personnel. The design of the Estelle compiler therefore was influenced by the need to get, at least, a subset Estelle compiler operating as soon as possible.

Other considerations, such as speed of the compilation or of the object program, are of minor importance at present. In the next section an overview of the compiler design is given.

4.2 Design Overview.

The design of the Estelle compiler differs radically from that of the first, familiarization attempt [vDi87]. While the first attempt will not be discussed in any detail, lessons learnt and how they effected the final design will be discussed. One of the most important lessons learnt was that it was senseless to "re-invent the wheel". A number of compilers of languages similar to Estelle were examined; for example, the original Pascal compiler [Wir71], the P4 Pascal compiler ([Pem82], [Pem82a]), the Edison compiler [Bri82] and the Pascal- compiler [Bri85]. The Pascal- compiler, which was chosen as the basis of the Estelle compiler, had the largest influence on the design.

While the theoretic model of compilation provides a clear division between different phases, in practice phases are often intertwined. A design decision therefore is how to organize the phases.

4.2.1 Organization.

In the first attempt, the Estelle Compiler was a single pass compiler, where all compilation phases, (i.e. lexical, syntactic and semantic analysis, and code generation) were organized as a single executable unit. Although this approach has distinct advantages, experience has shown that a multipass organization is more practical.

Advantages of single pass compilation include (1) speed of compilation and (2) simpler error-recovery during semantic analysis:

1. Because no intermediate files are generated, there is no disk accessing, which would seriously degrade speed of compilation.

2. In cases where semantic errors occur, and it is necessary to recover to a state in which the error does not continue to perpetuate, it should be possible to skip an entire construct, and if all phases are combined, this is possible. For example, in the expression
   $i_0 := c + i_1 + i_2 + i_3;$
   where i_j is an integer $(j = 0, 1, 2, 3)$ and c is a character, $c + i_1$ causes a semantic error, and every subsequent addition of i_j perpetuates the error, which would result in spurious error messages.

Disadvantages of single pass compilation include (1) the large memory requirements, (2) poor quality code generated and (3) difficulties with memory allocation and address resolution.

1. The memory requirement is largely due to the fact that the entire compiler is kept in memory, with all the necessary data structures to store information, (for example, the symbol table and in the case of a tabular parsing method, the parser tables), and blocks of intermediate code where memory allocation and address resolution remain to be performed.

2. Poor quality code is generated because without advance knowledge of code to be generated later, and with no look-ahead of future code, control flow code (e.g. for "goto") is impossible without storage of the generated code which requires address resolution.

3. As mentioned in (2) control flow code needing address resolution is difficult, Memory allocation is also difficult, because of the lack of look-ahead. The only viable solution is storage of code before address resolution and memory allocation is completed. This implies added complexity due to the necessity of keeping track of the location of instructions needing resolution.

Multipass compilation has the advantages of restricting memory requirements and resolving the problems of address resolution. In addition, it allows a clear decomposition of the compiler into modules. This contributes to the overall clarity, ease of implementation and the maintainability of the compiler.

Disadvantages of the multipass organization are (1) the degraded speed of compilation and (2) the necessity of transferring information between passes.

1. Speed of compilation is degraded because of the disk accessing needed to read and write temporary files for transferring information between passes.

2. The necessity of transferring information between passes results in useful information being lost, or needing to be transferred as well. For example, it may be necessary to transfer an entire symbol table, which can be large.

The organization of the Estelle Compiler is as follows:

Pass1: This pass contains the lexical analysis phase of the compiler, scanning the source text and producing an intermediate code file.

Pass2: Three distinct phases can be found in Pass2, namely syntactic analysis (parsing), semantic analysis, and intermediate code generation. Pass2 scans the intermediate code produced by Pass1, and in turn, produces an intermediate code file.

Pass3: Addresses are resolved by Pass3, which also performs simple code optimization. Pass3 reads the intermediate file produced in Pass2 and produces the final pseudo-code file.

Note that, because syntactic and semantic analysis, as well as intermediate code generation are contained in one pass, it is not necessary to transfer the symbol table between passes.

Another major design decision was the parsing method.

4.2.2 Parsing Method.

As has been mentioned, the initial design of the Estelle compiler incorporated a LL(1) based parser, while the method used in the final design is recursive-descent. A tabular method was chosen because it was felt that making the changes needed due to the changing language definition would be easier with a well documented and formal parsing method. LL(1) parsing was chosen primarily because its tables are smaller than for LR parsing and the simplicity of implementing a well understood LL(1) parser generator.

The reason for the change is that difficulties were experienced in code generation. In the LL(1)-based Estelle compiler described in van Dijk [vDi87], the use of semantic action routines called by the parser resulted in the necessity of explicit stacking of attributes prior to final code generation. By using recursive-descent parsing and integrating the semantic analysis and code generation into the parser, attributes are implicitly stacked by the implementation language's (Pascal's) activation records. This simplifies matters as the compiler does not need to keep track of the attributes. The integration of the semantic analysis and code generation phases with the recursive-descent parser allows the problems of semantic error propagation to be handled simply. Note that the difficulties experienced in code generation are not insoluble, but it was decided to pursue a method where such problems are well documented and where a number of compiler examples are available. This was not the case with LL(1)- based compilers, as most of the literature concerns deals exclusively with parsing and syntactic error recovery.

The decision to use recursive-descent was also taken in the interests of simplicity, an attitude backed by Tremblay and Sorenson [Tre85]:

> "A straightforward way of getting the job done is to use a recursive-descent parser."

Once the decision was taken to use recursive-descent parsing, decisions concerning semantic analysis and code generation involved only the design of the symbol table, and in case of code generation, also the target language.

4.2.3 The Symbol Table.

As performance of the system was not an overriding concern, and simplicity was desired as an aid in maintenance, a stack implemented list structured symbol table was decided upon. Actually, the symbol table is divided between two passes, namely Pass One and Pass Two. In Pass One, i.e. during lexical analysis, a hash based word table connected to a spelling table is used to distinguish between reserved words and identifiers, while also producing integer based unique indices for identifiers. Thus, in subsequent passes, a character string representing an identifier name can be uniquely identified by an integer, saving not only memory, but also resulting in faster access. In the list structured symbol table of Pass Two provision is made for different attributes to be stored by using variant records, while the stack implementation of blocks is used for scope analysis, as well as contributing to code generation by providing different levels for the relative address/level dynamic relocation memory scheme.

The target language to generate was another major decision.

4.2.4 Target Language.

When work was started on the Estelle compiler the FAPL meta- implementation was the guiding example of the type of system to be built, and consequently influenced the Estelle compiler's design. As the FAPL preprocessor compiles FAPL to PL/I [Poz82], the use of a high-level language as a target language was investigated [Kri86]. It was, however, decided to rather generate low-level code, which, together with a suitable run-time environment, would allow more control to be kept of the implementation, a necessity for the PDW. Specifically, pseudo-code is generated for a theoretical machine known as the **Estelle Machine**.

While the generation of pseudo-code, as opposed to native code, causes a degradation in the execution speed, the simplicity gained is significant.

Major problems inherent in code generation, such as instruction selection and register allocation are avoided, while other problems, such as memory management, are simplified. In addition, by rewriting the Estelle Machine, the system can be ported to another machine.

An Estelle machine which takes a pseudo-code program and interprets it, is the ideal environment for a meta- implementation. By placing another layer between the operating system and executing programs, the interpreter allows a protocol developer, validator or performance analyst to simulate the environment of the protocol. Another method used when a high-level language is used as the target language is to have a library of routines to link in after compilation producing a specific instance of the meta- implementation with certain characteristics. For example, an instance of a protocol is created with queues using a first- come-first-serve queuing discipline and no error demon. To produce a meta-implementation using a different queuing discipline or with an error demon creating errors, a recompilation or relink is needed. With the Estelle machine approach, run time changes can be made to, for example, queuing disciplines, error generation, carrying capacity or load factors. One can consider the approach similar to run time debugging. In fact the, pseudo-code debugger, a utility described in chapter 5, is a simple version of the type of program needed for this purpose.

It is interesting to note which approach was used by other implementors. Of the four known Estelle translators which have appeared in the literature recently, one, the NBS Estelle compiler, translates Estelle to C [Lin86], another reported by de Souza and Ferneda translates to Pascal [dSo88], while one reported by Ansart et al. translates to an internal representation [Ans87]. The ESTIM Estelle interpreter of de Saqui-Sanni and Courtiat [dSa87] makes use of a functional language as an intermediate code form which can be interpreted. The approach used by the NBS, as well as de Souza and Ferneda, is the same as that used for FAPL, namely generation of high-level code, compilation of the resulting program, together with additional routines, by a compiler for the target language. The approach used by Ansart et al. and de Saqui-Sanni and Courtiat is similar to the pseudo-code approach as the internal representation is used as the input for components of a PDW-type protocol development environment.

In the next chapter the development of the compiler system will be discussed.

Chapter 5

Implementation Notes.

The development of a compiler involves three major processes, namely specification, design and implementation [Hor74a]. The design of the Estelle compiler has been discussed in chapter 4., together with the issues effecting the design, i.e. the specification. The process of implementation will now be discussed. Issues include the implementation strategy, testing strategy, software standards and software development tools.

5.1 Implementation Strategy.

While the design of the compiler has been discussed, the process of design is important as the design strategy is entwined with the implementation. The design/implementation strategy used was top-down design and bottom-up implementation.

The top-down design strategy defines that the design decisions should be identified, ranked in order of importance and used as the basis of the design. Important design decisions, i.e. those decisions which effect the largest possible portion of the design, and restrict further decisions as little as possible, are addressed first in the design process [Gil83]. Stepwise - refinement, a form of top-down design used for the development of the Estelle compiler, consists of the refinement of a program and data specification in a series of refinement steps eventually getting to the code level of definition [Wir71a].

The concept of a module and modularization of a program is an important concept in software development. There are a number of approaches to decomposing programs into modules. For example, a module can be de-

veloped as a collection of procedures which together provide a major step in the processing. In the Estelle compiler the "information-hiding" [Gil83] (See also [Par72]) approach was preferred. That is, modules are built around major design decisions, and access by other modules is controlled. Where possible, the modules of Estelle where designed around data structures, or variables. For example, the module "*ScopeAnalysis*" in Pass Two defines the symbol table manager, that is, the symbol table itself and the procedures which modify it. Although this is the preferred approach, practicalities dictate a policy of pragmatism. To summarize the major modules in the Estelle compiler and the decomposition approaches:

For decomposition into the three passes, namely Pass One, Pass Two and Pass Three, the decomposition is based on information hiding. As syntactic and semantic analysis and intermediate code generation all make use of the symbol table, they should be grouped together. As lexical analysis and address resolution and code optimization do not, they are separated.

A second level decomposition is the division of Pass One into the modules "*Administration*", "*Input*", "*Output*", "*Word – symbols and Identifiers*" and "*Lexical Analysis*". While "*Word – symbols and Identifiers*", which defines the word table manager, and the modules "*Administration*", "*Input*" and "*Output*", which define the interface with the outside world, are decompositions based on information-hiding, the module "*Lexical Analysis*" is a process-orientated module.

While strict modularization requires that a well-defined inter-module interface should be defined, as is the case for the modules Pass One, Pass Two and Pass Three, (where the intermediate code files define the interface), practicalities once more dictate a more lenient policy. For example, if a strict interface was defined for the symbol table in Pass Two, i.e. not only would structural changes be made to the symbol table via the procedures of the symbol table manager, but also the insertion, changing and interrogation of information. This would cause a terrible bottle-neck, as the symbol table is the only major data structure in the compiler, and is accessed continually during compilation.

An aspect of the design which has not been addressed is the use of a standard design. While the design of the Estelle compiler can be rationalized in terms of the design strategy used by the author, the influence of other compilers is not denied. In fact, the design of the Estelle compiler closely resembles that of the Pascal- compiler described by Brinch Hansen [Bri85]. The use of the design of an existing compiler saves time and avoids pitfalls [Hor74a]. The Pascal- compiler was actually used as the basis of the

implementation of the Estelle compiler.

In the implementation of the Estelle compiler the procedural level is used as a cut off level to halt further stepwise- refinement of the design. I.e. when individual procedures have been identified, bottom-up implementation is started. Bottom-up implementation implies that the lowest level procedures are implemented first, and the procedures making use of them are implemented next, and so forth, until the implementation is built up from the bottom up to the highest level procedure, i.e. the program itself. This bottom-up implementation of procedures is associated with the testing strategy used, which will be discussed in section 5.2.

Another aspect in the implementation of the compiler, was the use of existing algorithms, for example, from Aho et al. [Aho83], Knuth [Knu73] and so forth. By using existing, published algorithms with known properties, the likelihood of producing a correct compiler is increased. Naturally, the use of off-the-shelf-components, i.e. hacking out portions of existing compilers and using them, is a similar idea with similar advantages. This strategy was also used in the implementation of the Estelle compiler, as the Pascal-compiler was used in its entirety as the basis of the present implementation. In addition, while not using formal programming ([Dij76], [Gri81]) directly, the philosophy of this type of program development influenced the implementation.

The testing strategy for the implementation, which will be discussed next, directly concerns the implementation plan of the compiler.

5.2 Testing Strategy.

Testing is an important aspect of any implementation, dictating whether the implementation is acceptable. As has been mentioned, the testing strategy used for the Estelle compiler is associated with bottom-up implementation.

As each component of the compiler was completed, it was individually tested, after which further tests were performed when the component was integrated with other components of the system. For example, in each pass various procedures were tested, after which the entire pass was tested. Thus, the testing and implementation strategies of the Estelle compiler combined to form the cycle implement, test, integrate, test, implement, and so forth. The testing could be divided into two components, namely ongoing testing and acceptance testing.

Ongoing testing was performed while implementation was taking place,

i.e. every time a procedure was added it was tested, while acceptance testing implies a rigorous testing effort at a predefined implementation check-point. Brinch Hansen [Bri85] defines a number of rules for testing compilers:

1. A compiler is tested by compiling small programs specifically constructed for testing.
2. The compiler must display the test program character by character as it is read in.
3. The test output must consist of the contents of the intermediate code file.
4. The test program must be written in such a way that it forces each statement of the compiler to be executed at least once.
5. When the compiler uses a variable restricted to a subrange of values, the test should cover the entire range of values (or at least the extreme values of the range).

The software tools "*Tokens*", "*Pretty*" and "*Debug*" are used in testing, but will be described later. The implementation plan of the Estelle compiler system was as follows: (Where implementation of a component implies that ongoing testing has been performed.)

5.3 Implementation Plan.

The implementation called for a number of implementation phases:

5.3.1 Phase 1. Re-implementation of the Pascal- compiler.

During this re-implementation of the Pascal- compiler modifications were made to the existing design. While the changes were minor, they were necessary, as the Pascal- compiler was defined in the language Pascal+. (See Brinch Hansen [Bri85].)

Each of the Pascal- compiler's three passes, as well as the Pascal- pseudo code interpreter, was implemented and then tested. Testing took place using the test programs constructed by Brinch Hansen for the Pascal- compiler from the test rules above (See Brinch Hansen [Bri85]).

5.3.2 Phase 2. Implementation of the Pascal Subset.

During this phase the Pascal- compiler was modified and extended until the full Pascal subset of Estelle had been implemented. Each pass and the code interpreter were implemented and tested separately. The test programs for the three passes and the code interpreter were extensions of the test programs of the previous pass.

Note that some of the features of the Pascal subset were not been adequately tested as they make use of heap memory management routines which are part of the run time environment which was yet to be implemented. For example, "**new**(p)", where p is a pointer variable of some type. In addition, Appendix D contains a list of features which were left for later implementation.

5.3.3 Phase 3. Implementation of Estelle.

The last phase consists of the complete implementation of the Estelle. As in the previous phase, each pass was modified and tested separately.

As was mentioned before, as the run time environment has not been implemented, the correctness of the compiler cannot be proved. However, it can be said that the compiler is consistent as it produces the same output for the same test program. Whether pseudo code and the accompanying interpreter allows an Estelle specification to be run, must be tested once the run time environment has been completed.

In the next section software standards used in the implementation are discussed.

5.4 Software Standards.

During the implementation of the Estelle compiler a number of rules were used to define software standards, namely:

1. The use of nonstandard features of the implementation language must be avoided. However, in a number of cases this rule had to be ignored. For example, the type "word" in Microsoft Pascal is a nonstandard Pascal data type, but was used in the compiler for the data representation for sets, as packed arrays of bits are not implemented.

2. Indentation and format of the compiler text must be standard throughout, making the text easier to read.

3. Nomenclature of procedures, functions, conceptual modules, types, constants and variables must be as standard as possible, and must be informative. Note that in Pass Two procedure names are exactly the same as in the grammar rules, and in Pass Three the same for code rules.

4. Internal documentation may be sparse, as the above rule ensures that internal documentation is provided by the source text itself. In Pass Two and Pass Three comments immediately prior to a procedure definition contain the syntax and code rules processed by the procedure. Conceptual modules in the entire compiler system are documented with comments to indicate the extent of the conceptual module.

5. Structured programming must be used throughout. That is, algorithms used in the implementation must make exclusive use of structured programming constructs such as while, for, repeat, if and case statements. Goto statements must be avoided.

6. The source text must be divided into conceptual modules reflecting the design of the compiler. This last rule ensures that the procedures of, for example, the symbol table manager, are topologically near, allowing the various functions performed on the common data structure to be examined together.

In addition to the above rules, note that the use of standard algorithms and off-the-shelf components have been used as a policy of attempting to ensure correctness of the compiler system.

Tools used during the implementation and testing of the Estelle compiler are discussed in the next section.

5.5 Software Development Tools.

A number of tools were developed and used during the implementation of the compiler system. The tools developed include "*Xref*", "*Tokens*", "*Debug*", "*Pretty*" and "*Debugger*":

5.5.1 "*Xref*".

"*Xref*" is a cross reference generator and program lister modified from one described by Grogono [Gro80]. The program is called with the source file,

target file and the number of the configuration file as parameters. The configuration file contains the name of the file containing standard words, the number of standard words, language version, i.e. Estelle or Pascal, three booleans indicating whether standard words should be included in the cross reference listing, whether the source text should be printed and whether a cross reference listing should be printed, the number of lines per page, number of characters per line and number of lines between entries.

When cross referencing a Pascal program or an Estelle specification standard words are often not wanted in the cross reference listing, and therefore *"Xref"* provides for this case. In addition, a source listing or cross reference listing may not be wanted, which is also provided for. The parameter for the version is used to print the language version in the header of each page of the source and cross reference listing. The number of lines, characters per line and number of lines between entries are used to define the format of the listings for a specific output medium. The following is a typical configuration file (Say configu1.dat):

```
pascal.std
77
1
false
true
true
60
79
0
(* File for standard words *)
(* Number of standard words+1 *)
(* Version: 0 =estelle, 1 =pascal *)
(* Print standard words *)
(* Print source listing *)
(* Print cross reference *)
(* Number of Lines per page *)
(* Number of characters per line *)
(* Number of lines between entries *)
```

"Xref" prints the first line of a file after the header on each page. This, together with the date and time stamp printed in the header is useful in keeping track of development files.

5.5.2 "*Tokens*".

"*Tokens*" is used to print a file of tokens in a readable fashion. Because a token file is a sequence of integers and reals with some of the integers representing the tokens and others the tokens' arguments, it is difficult to verify that the tokens are correct. "*Tokens*" reads a token file and prints the name of a token together with its argument if it has one.
For example, the token sequence
"58 1 58 2 88 42 37"
will be printed as
"*NewLine*(1) *NewLine*(2) *Specification Identifier*(37)".

5.5.3 "*Debug*".

"*Debug*" is not a separate program, but actually a module which can be linked to Pass Two of the compiler to provide debugging code. The module provides the dumping of symbol table entries in a readable fashion, i.e. the names of fields, as well as their contents, are printed. When a field is a pointer to another entry in the symbol table, that entry is also dumped. "*Debug*" also contains the active routines from "*Tokens*" and "*Pretty*", thus allowing a trace to made of the input, output (provided by the routines from "*Pretty*") and symbol table entries.

5.5.4 "*Pretty*".

"*Pretty*" is very similar to "*Tokens*", as it prints the intermediate code files generated by Pass Two and Pass Three in a readable fashion. The pseudo code operation is printed, followed by its parameters. "*Pretty*" therefore aids in the debugging of Pass Two as well as Pass Three.

5.5.5 "*Debugger*".

"*Debugger*" can be seen as an extension of the Estelle machine. Routines, such as a stack dump, were added to the Estelle machine to provide for debugging of the Estelle machine and also to evaluate the code generated for a specification. In addition, "*Debugger*" contains some of the routines needed for a meta-implementation. The active routines from "*Pretty*" provide for a trace of instruction execution. Routines to collect statistics on instruction frequency provide information indicating which instructions should be optimized. A command line interpreter (CLI) allows commands to be given

to examine and modify stack values, registers values and to step through instructions of a loaded specification. Commands can also be constructed which specify E-code instructions which should be executed. Appendix F contains a specification of the debugger.

Further developments will allow the examination and modification of process information, interaction point binding, queue contents and other information concerning the module/process hierarchy. In this respect, the ESTIM Estelle interpreter system [dSa87] provides an exellent model of the type of system needed.

In the next chapter Pass One's design and implementation will be discussed.

Chapter 6

Pass One.

The draft international standard [ISO87] provides a specification of a lexical analyzer for the Pascal subset of Estelle. Appendix A contains a specification of a lexical analyzer for Estelle based upon this. The design of the lexical analyzer is discussed first, after which the implementation is discussed.

6.1 The Design of the Lexical Analyzer.

The design of the lexical analyzer is described using augmented Finite State Machines (in fact Mealy machines). The lexical analyzer sends recognized tokens and their arguments to an intermediate code file. These tokens and their arguments are indicated in the diagrams as the output associated with the transitions of the Mealy machine. The various parts of the lexical analyzer are described separately, and unnecessary detail is omitted: (Figure 6.1. provides an overview of the lexical analyzer.)

The states used to collect identifiers also collect word- symbols and directives. This follows because the definitions of identifiers, word-symbols and directives are all *Letter* { *Digit* | *Letter* }. The character-sequence must first be collected before the token can be identified as an identifier, word-symbol or directive. In a similar fashion, the states used to collect numbers (reals or integers), also collect labels. This follows as a label's definition is the same as a Unsigned-integer, namely *Digit* { *Digit* }.

The states to collect identifiers, word-symbols and directives are the first to be described.

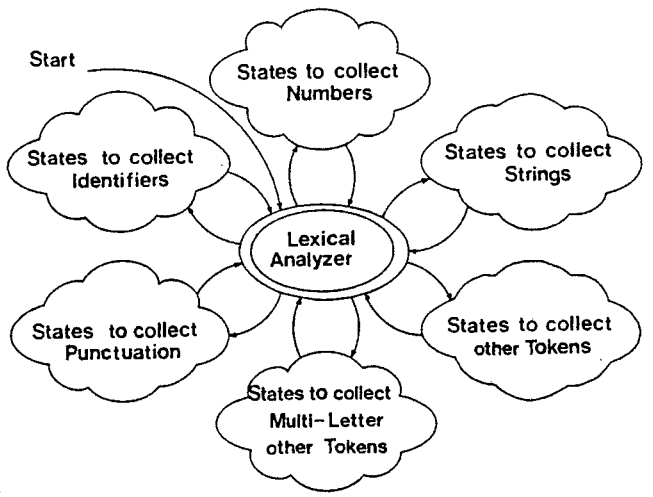

Figure 6.1: An Overview of the Lexical Analyzer.

6.1.1 Identifiers, Word-symbols and Directives.

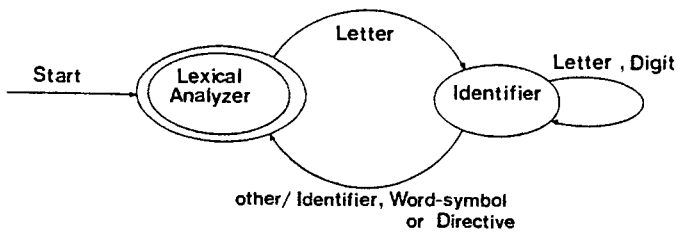

Figure 6.2: Identifiers, Word-symbols, Directives.

Once a string of characters has been collected, screening must take place to decide whether the collected string of characters is a reserved word, a directive or an identifier. A symbol table, or rather a word table is used to distinguish between the various tokens, allowing a unique value, specifying an index to a string of characters in the symbol table in the case of an identifier, or, in the case a reserved word, i.e. a word-symbol, directive or standard identifier, the ordinal value of its representation. Thus, either the token "*Identifier*" followed by a value, or only the ordinal value of a reserved word's internal representation, is written to the intermediate code file.

Note that a restriction is placed on the length of an identifier, allowing only identifiers with lengths not exceeding the length of a line, i.e. a carriage return or line-feed character terminates an identifier.

It is fruitful to discuss the design of the symbol table itself. As can be seen in Figure 6.3., the symbol table consists of a hash table, word lists and a spelling table.

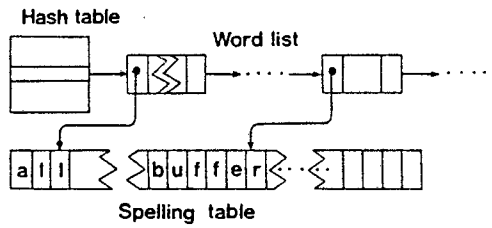

Figure 6.3: The Word Table.

The spelling table is used to store unique sequences of characters which represent word-symbols, directives and identifiers, while the word lists contain lists of words with the same hash key in the hash table. The word lists consist of linked lists of word records, which each define a word-symbol, directive or identifier. Fields in the record contain the length of the word-symbol, directive or identifier, indicate whether it is an identifier or not, indicate the last character of the word-symbol, directive or identifier in the spelling table and a field which contains an identifier index or the ordinal value of the word-symbol, directive or standard identifier's representation in the lexical analyzer (scanner).

The string of characters which has been collected is used as the argument of a hashing function to provides an index to the hash table. The word list is searched to find whether the word already occurs. If not, the string of text is inserted into the spelling table, a word record defining it is setup and placed in the associated word list of the hash- table.

Note that the method used is open hashing, with direct chaining as the collision resolution method. (See section 3.3.)

The next class of tokens to be collected is that concerned with numbers.

6.1.2 Integers, Reals and Labels.

Note that leading signs (i.e. unary plus or minus), are not associated with a number, the parser identifies this sequence of tokens. Note also that the state "*Real2*" must have either a "+" or a "−" or a digit as input,

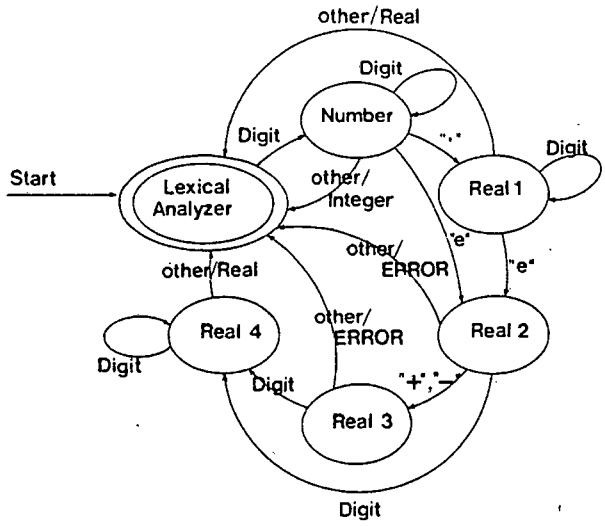

Figure 6.4: Integers, Reals, Labels.

otherwise an error results. Figure 6.4. does not show the details of other error conditions that might occur, such as under/overflow of integers, reals or a real's exponent part, but such error conditions should be assumed.

The tokens sent to the intermediate code file are "*Integer*" followed by an integer value for integers and labels and "*Real*" followed by a real value for reals.

Restriction: The size of integers, reals and exponents are restricted to values representable in the implementation language.

Character-strings and character-constants are now examined.

6.1.3 Character strings and Character constants.

Note that no distinction is made between character strings and character constants and that when a character string is interrupted by the end of text character, an error is reported. Character strings are sent to the intermediate code file as a sequence of the token "*Graphic*" followed by the ordinal value of the next character in the string, separated by the token "*Comma*". For example, 'AB' would be sent to the intermediate file as
"*Graphic*" ord(A) "*Comma*" "*Graphic*" ord(B).

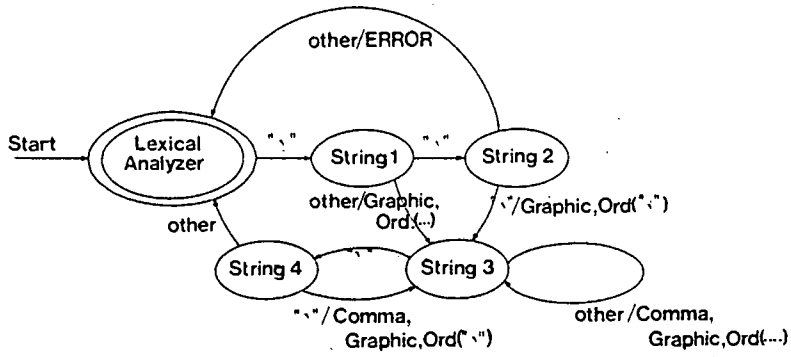

Figure 6.5: Character strings, Character constants.

Restriction: Character strings and character constants are not distinguished. This is to simplify matters, as a character constant is a character string of length one.

There are tokens formed from sequences of characters other than those mentioned previously:

6.1.4 Other Tokens (more than one character).

Notice that on finding a comment, the lexical analyzer reverts to the start state without specifying a token. Comments are handled by a recursive procedure outside the main lexical analyzer.

Some tokens consist of only one character:

6.1.5 Other Tokens (one character).

For these tokens, the character itself is used as an index to an element in an array which contains the token associated with the character. For example, the character "(" is associated with the element "$SpecialSy['(']$" which contains the token "$LeftParenthesis$".

Some characters cannot occur legally outside an apostrophe- image:

6.1.6 Illegal Characters.

The token "$Unknown$" is sent to the intermediate code file. Illegal characters are handled in a similar fashion to the one character "other" tokens. An error is reported indicating where the illegal character was found.

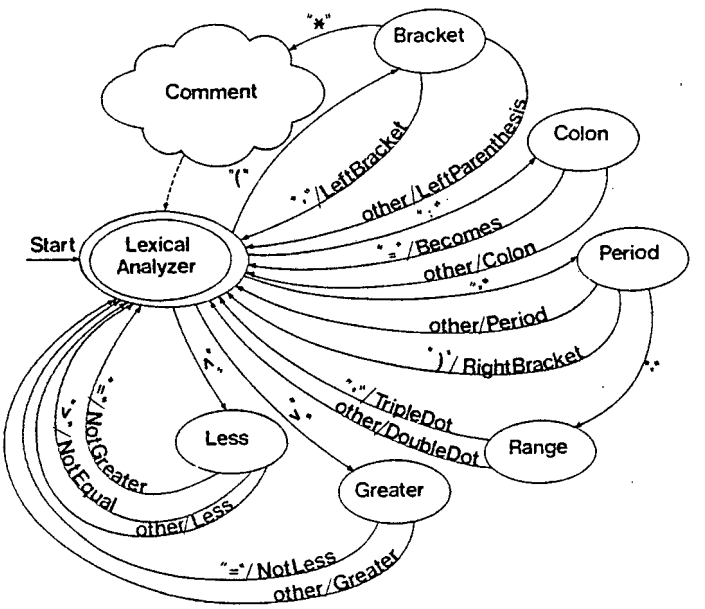

Figure 6.6: Other Tokens (more than one character).

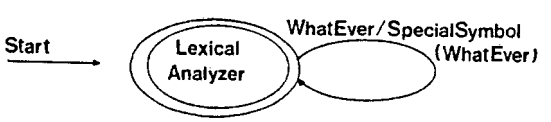

Figure 6.7: Other Tokens (one character).

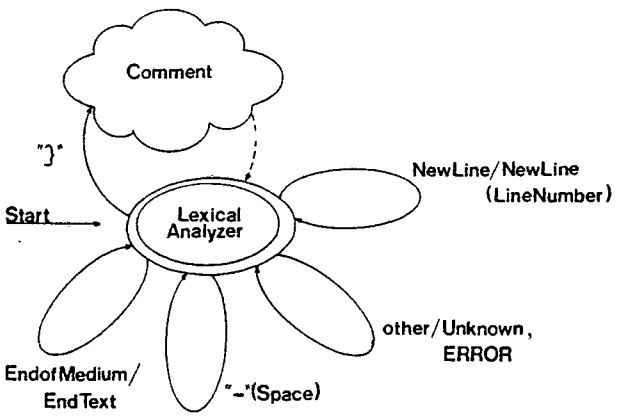

Figure 6.8: Punctuation.

Certain characters are used to separate tokens:

6.1.7 Punctuation.

Notice that the comment state reverts to the start state without producing output. When a carriage return or line feed is encountered the token "*NewLine*" followed by an argument, the new line number, is sent to the intermediate file. This is used in Pass Two to print line numbers for error messages. In a similar way, the token "*EndText*" is sent to the intermediate file when the end of file character is encountered from the initial state. If the end of the text is encountered when the scanner is in the comment state, an error is reported.

The following section describes the implementation of the above design. Constants, types, variables, procedures and functions described are those that occur in the source text of Pass One.

6.2 The Implementation.

Before discussing the main block of Pass One, or, conceptually, the main module, a number of modules which provide services used by the main module will be discussed.

6.2.1 Service Modules.

The lexical analyzer interfaces with four service modules, "*Administration*", "*Input*", "*Output*" and "*Word – symbols and Identifiers*". The externally visible behaviour of the modules will be described next, with some additional information concerning internal behaviour where applicable.

6.2.1.1. "*Administration*".

"*Administration*" presents the services "*Emit*", "*EmitReal*", "*NewLine*", "*Error*", "*TestLimit*", "*StartMessage*" and "*FinalMessage*". "*Emit*" provides the service of writing a value to the intermediate code file. In the same way, "*EmitReal*" provides the service of writing a real value to the intermediate code file. "*NewLine*" updates the current line number used for error reporting, as well as setting a boolean, "*CorrectLine*", which is used to suppress error messages for line numbers that have already occurred in error messages. This error suppression was discussed in section 3.1.4.

Errors in the source text are reported by calling "*Error*" with a parameter of the type "*ErrorKind*", an enumeration type. A list of lexical errors can be found in Appendix C. "*Error*" sets the boolean "*Emitting*", which specifies that "*Emit*" and "*EmitReal*" must not send output to the intermediate code file. This speeds up the pass once an error has occurred.

"*TestLimit*" is used to test whether any limits are exceeded, for example, if the string table is full. If a limit was exceeded, an internal compiler error is reported, and compilation is aborted.

"*StartMessage*" is used to give positive feed back when execution starts (See section 3.1.4.). For example, the compiler name, pass number, and compiler version is identified. "*FinalMessage*" is used for feed back when lexical analysis has been completed. The fact that the pass has been completed is reported, as well as the lack of errors, if that is the case.

6.2.1.2. "*Input*".

"*Input*" provides the service "*NextChar*", which gets the next character from the input file, skips invisible characters, (i.e. characters which are not in the range ASCII 32 to ASCII 126), as well as handling the end of file and end of line conditions. When an end of file condition is encountered, the character returned is "*ETX*" (end of text), while end of line results in a "*Nl*" (new line) character being returned.

6.2.1.3. "*Output*".

"*Output*" provides the services "*Emit1*" and "*Emit2*" which are respectively used to output a token, and a token with an argument via "*Emit*". As is the case with input, where a procedure is called each time a character is read from the source file, the use of procedures for writing tokens and arguments

to the file slows down the scanner. However, this provision of a uniform input interface is a sensible programming practice.

6.2.1.3. *"Word – symbols and Identifiers"*.

This service module provides the services needed for the screening of word-symbols, directives and identifiers. That is, it is the physical implementation of the word table discussed in section 6.1.1. Naturally, all the word-symbols and directives, as well as all standard identifiers for standard constants, types, procedures and functions are predefined.

For this module it is fruitful to examine the data structures and the operations there upon in more detail. The following is the definition of the spelling table:

```
const
   MaxChar = 5000;
type
   SpellingTable = array[1..MaxChar] of char;
```

A variable, *"Characters"*, defines the number of characters stored in the spelling table. The word lists are implemented as linked lists of word records, which each define a word- symbol, directive or identifier. A word record is defined by the following definitions:

```
type
   WordPointer = ↑ WordRecord;
   WordRecord =
      record
         NextWord: WordPointer;
         IsIdentifier: boolean;
         Index, Length, LastChar: integer
      end;
```

The hash table itself consists of an array containing the word lists associated with the indices of the array:

```
const
   MaxKey = 631;
type
   HashTable = array[1..MaxKey] of WordPointer;
```

The procedures "*Insert*", "*Define*", "*Search*", "*Found*", and "*Key*" are operations defined on the word table. "*Key*" is the hashing function which can be applied to a string of text, i.e. word-symbol, directive or identifier. "*Insert*" inserts the string of text into the spelling table, sets up a word record defining it, and places it in a word list of the hash table. "*Define*" controls both "*Insert*" and "*Key*" to place a new string in the word table. "*Found*" ascertains whether a string is in the word table, while "*Search*" searches for a string, and inserts it, if it was not found. Each time search adds an entry into the word table, the variable "*Identifiers*", which contains a count of the number of identifiers, is incremented. In this way, unique values are assigned as the string's identification.

Only "*Search*" and "*Define*" are externally visible services presented by the module.

The word table is initialized by the the procedure "*Initialize*", which predefines all word-symbols, directives and standard identifiers of Estelle. For each of the word- symbols and directives "*Define*" is called with a parameter indicating that the string being defined is not an identifier, as well as a parameter indicating the ordinal value of the enumeration type used to represent the token associated with that directive or word-symbol. "*Define*" is used in a similar way to define the standard identifiers, as it is called with a parameter indicating that an identifier is being defined. The parameter which associates the string with a unique integer has a constant value indicating which standard identifier it is. These constants are the same in Pass Two and allows Pass Two to identify standard identifiers. After all the word-symbols, directives and standard identifiers have been predefined, the variable "*Identifiers*" contains the next integer which can be used.

In the next section the main module of Pass One, "*Lexical Analysis*", is discussed.

6.2.2 "*Lexical Analysis*".

"*Lexical Analysis*" is a physical implementation of the Mealy machine discussed in section 6.1. The module contains four procedures, "*BeginLine*", "*EndLine*", "*Comment*" and "*NextSymbol*". "*BeginLine*" and "*EndLine*" are procedures to update the present line number "*LineNo*", which is used in error reporting, as well as to write a token/argument pair to the intermediate file. This token/argument pair, "*NewLine*" followed by the line number, is used to update the line number in Pass Two. "*Comment*" is a procedure used to recursively screen out

comments from the source text. Note that this use of recursion is not visible in the Mealy machine for punctuation (Figure 6.8.).

One procedure call of the procedure "*NextSymbol*" corresponds to a transition to one of the collections for states in Figure 6.1., i.e. the states to collect to one of the token types described in sections 6.1.1. to 6.1.7. The "*NextSymbol*" procedure consists of a case-statement with case constant lists corresponding to each of the token types. For example, the states used to collect numbers are physically implemented in the statements which are selected by the case constant list '0'..'9'. By removing the statements concerning token separators from the case-statement, and performing it in a while loop before the case statement, performance is improved. The statements for token separators produce no tokens, and it is wasteful to have a number of procedure calls to remove a number of spaces from the source text.

Apart from a number of sets, such as "*CapitalLetters*", "*Digits*", and "*AlphaNumeric*", used to differentiate between classes of characters and the data structures used for the word table, only one data structure can be identified in Pass One, namely the array variable "*SpecialSy*" which is used to convert single character special symbols to tokens (See section 6.1.5.).

The enumeration type "*SymbolType*" defines the representation of tokens in Pass One, in the intermediate code file, and in Pass Two. The internal representation of a token is sent to the intermediate code file via the procedures "*Emit1*" and "*Emit2*", from where Pass Two will read the tokens. The code file is therefore a sequence of integers, representing tokens, followed in some cases by integer or real values which are the arguments of the token.

The main program consists of a call to "*Administration*", the initializing procedure of the module "*Administration*", as well as a procedure call to "*Initialize*", which is used for initializing the word table, as has been mentioned, as well as being used to initialize the array "*SpecialSy*" and the sets used to distinguish character classes. Additionally, a token pair is sent to the intermediate file indicating that the first line of source is about to be scanned and the first character is read from the source file. The next part of the main program is a while-loop which indicates that while the end of text character has not been encountered, "*NextSymbol*" should be called, after which the "*EndText1*" is sent to the intermediate file. Finally, "*FinalMessage*" is called.

In the next chapter Pass Two of the Estelle compiler is discussed.

Chapter 7

Pass Two.

This pass consists of the syntactic and semantic analysis phases of the compiler, as well as intermediate code generation.

Although the processes of syntactic, scope and semantic analysis, as well as intermediate code generation are entwined in Pass Two, the design of each will be discussed separately. However, an overall picture is needed first.

7.1 The Design of Pass Two.

Pass Two reads the tokens and arguments from the intermediate file produced by Pass1, checks that the syntax conforms to that of an Estelle specification, (i.e performs syntactic analysis), checks the semantics of the syntactic structures recognized, (i.e. performs scope and type analysis), and finally generates preliminary pseudo code for the semantically correct syntactic structures. The design of the syntactic analyzer will be discussed first.

7.1.1 Syntactic Analysis.

The Estelle compiler's recursive-descent parser was constructed using the construction rules presented by Brinch Hansen [Bri85]. For example,

1. For every BNF (Backus-Naur Form) rule $N \rightarrow E$, a procedure "N" is defined as:

   procedure N;
   begin

$a(E)$
end;

Where $a(E)$ is a parsing algorithm which recognizes the syntactic structure E.

2. A sequence of sentences of the forms $F_1, F_2, \ldots, F_n$ is recognized by parsing the individual sentences one at a time in the order written:

   $a(F_1, F_2, \ldots, F_n) \equiv a(F_1);\ a(F_2);\ \cdots;\ a(F_n);$

   Further rules are given for the construction of procedures to recognize a single symbol s, a BNF rule N, a possibly empty sentence $[\ E\ ]$, a sentence of the form $\{\ E\ \}$, a sentence of the form $T_1|T_2|\cdots|T_n$, where $T_1, T_2, \ldots, T_n$ are all not empty, as well as a sentence of the form $\{\ E\ \}^+$. These construction rules will be discussed further when error recovery is added to the parsing algorithm.

   Because recursive-descent parsers need a LL(1) grammar, certain restrictions must be placed on the grammar to make it conform to the requirements. The restrictions placed on the grammar to allow recursive-descent parsing correspond to those placed on LL(1) grammars. These restrictions result because the parser reaches points in the specification text where several different kinds of sentences may occur. The parser must decide which of these possible sentences to pursue. To avoid the necessity of back-tracking, the grammar of the programming language must be designed to allow the parser to make this choice by looking at the next symbol only. Brinch Hansen [Bri85] states the restrictions as follows:

   (a) For all sentences of the form $N \rightarrow E|F$, the following condition must be true:

   $First(E) \cap First(F) = \emptyset.$

   I.e. the intersection of the first sets must be empty.

   (b) If some of the sentences described by a BNF rule N may be empty, then the following must be true:

   $First(N) \cap Follow(N) = \emptyset.$

Where $First(\alpha)$ is the set of first symbols of the string α, while $Follow(\alpha)$ is the set of all the first symbols of all the constructs which can follow α. A formal definition can be found in the general literature ([Aho72], [Tre85], [Aho86] and Brinch Hansen [Bri85]) provides a simple method of finding first and follow sets.

Notice however, that rule (1) is often not true as many different types of sentences in Estelle begin with the symbol "*Identifier*". For example, assignments, procedure calls, etc. This sad state of affairs can be solved by either (1) using semantic information to distinguish the identifier concerned as a variable or a procedure identifier, or (2) modifying the grammar so that no choice needs to be made.

The latter method is known as left-factoring, as the left factor common to the sentences of that form is placed in a separate sentence, making it a unique first symbol of that point in the grammar. The next symbol of each sentence is now used to distinguish between sentences. For example, the BNF rule $N \rightarrow xa\alpha|xb\beta$ demonstrates this problem. The parser can not distinguish which of the two sentences it must pursue, as the symbol x is the first symbol of both sentences. The grammar can then be modified by rewriting the rule as follows: $N \rightarrow xN_1$, $N_1 \rightarrow a\alpha|b\beta$, where α and β are strings of terminals and nonterminals.

This method is, however, not always successful. In some cases only semantic information can distinguish between sentences. For example, in Estelle the assignment statement can have the form $Variable := Expression$, where $Variable$ is any type of variable excluding module variables, or $ModuleVariable := ModuleVariable$. Because there is syntactically no difference in the assignment $a := b$ for module variables and for other variable types, an assignment $a := b + c$ is syntactically correct for all variable types excluding module variables. Semantic information can be used to distinguish which sentence to pursue. I.e. once the variable a is found to be a module variable, the sentence $ModuleVariable := ModuleVariable$ is pursued.

Recursion is used in the recursive-descent parser when the grammar contains recursive references. It should be noted that the rules written with left-recursive references can cause problems and should be avoided, For example, the rule $Expr \equiv Term\ [\ Oper\ Expr\ ]$ (right-recursive) can be written as: $Expr \equiv [\ Expr\ Oper\ ]\ Term$ (left-recursive). The use of the second, left-recursive rule as the basis for constructing an algorithm would result in the algorithm recursively

calling itself on the first symbol, and therefore causing failure of the parser. Notice however, that this grammar rule does not conform to the grammar restrictions.

The error recovery method used in the Estelle compiler is the standard method mentioned in section 3.1.4. The effectiveness of the method depends on the choice of the set of synchronizing symbols. The heuristic used in the Estelle Compiler is based on using $Follow(A)$ for the synchronizing set of the nonterminal A. The symbol "*EndText*", which indicates the end of the specification text, is included in all the synchronizing sets. Additionally, extra symbols can be added to the synchronizing set within a sentence to closely specify error recovery in such a sentence.

The construction rules discussed above, and modified to accommodate the error-recovery mechanism, follow: (From Brinch Hansen [Bri85])

(a) For every BNF (Backus-Naur Form) rule $N \rightarrow E$, a procedure "N" is defined as: (Where $a(E, Stop)$ defines an algorithm which recognizes E. If an error occurs, the algorithm will report an error and discard symbols until the next symbol is in the synchronizing set "*Stop*".)

procedure $N(Stop : Symbols)$;
begin
  $a(E, Stop)$
end;

(b) A sequence of sentences of the forms $F_1F_2 \cdots F_n$ followed by a stop symbol is recognized by parsing the individual sentences one at a time in the order written:

$$\begin{aligned} a(F_1F_2 \cdots F_n, Stop) \equiv\ & a(F_1, First(F_2F_3 \cdots F_n) + Stop); \\ & a(F_2, First(F_3F_4 \cdots F_n) + Stop); \\ & \quad \vdots \\ & a(F_n, Stop) \end{aligned}$$

(c) When the parser expects a single symbol s followed by a stop symbol, it calls a procedure "*Expect*": ("*Expect*" will be discussed later.)

$$a(s, Stop) \equiv Expect(s, Stop)$$

(d) To recognize a sentence described by a BNF rule named N the parser calls the corresponding procedure named N using the stop symbols of the sentence as a parameter:

$$a(N, Stop) \equiv N(Stop)$$

(e) The parser uses the following algorithm to recognize a sentence of the form [E] followed by a stop symbol: (The procedure "*SyntaxCheck*" will be defined later.)

$$a([\ E\ ], Stop) \equiv SyntaxCheck(First(E + Stop));$$
$$\textbf{if } Symbol \textbf{ in } First(E) \textbf{ then } a(E, Stop)$$

(f) A sentence of the form { E }, followed by a stop symbol is recognized by the following algorithm:

$$a(\{\ E\ \}, Stop) \equiv SyntaxCheck(First(E) + Stop);$$
$$\textbf{while } Symbol \textbf{ in } First(E) \textbf{ do}$$
$$a(E, First(E) + Stop))$$

(g) If all sentences of the forms $T_1, T_2, \ldots, T_n$ are nonempty, the following algorithm will recognize a sentence of the form $T_1|T_2|\cdots|T_n$ followed by a stop symbol: ("*SyntaxError*" will be defined and discussed later.)

$$a(T_1|T_2|\cdots|T_n, Stop) \equiv \textbf{if } Symbol \textbf{ in } First(T_1) \textbf{ then}$$
$$a(T_1, Stop)$$
$$\textbf{else if } Symbol \textbf{ in } First(T_2) \textbf{ then}$$
$$a(T_2, Stop)$$
$$\vdots$$
$$\textbf{else if } Symbol \textbf{ in } First(T_n) \textbf{ then}$$
$$a(T_n, Stop)$$
$$\textbf{else } SyntaxError(Stop)$$

Note that if any of the sentences may be empty, "*SyntaxError*" is replaced by "$SyntaxCheck(Stop)$".

(h) A sentence of the form { E }$^+$ can be written as E { E }, and { E }$^+$ followed by a stop symbol can thus be recognized by the following algorithm:

$$a(\{\ E\ \}^+, Stop) \equiv a(E, First(E) + Stop);$$
$$\textbf{while } Symbol \textbf{ in } First(E) \textbf{ do}$$
$$a(E, First(E) + Stop)$$

Note that a variation of rule (h) is: The following algorithm recognizes one or more sentences of the form E separated by the symbol s, and followed by a stop symbol:

$$a(E\{\ sE\ \}, Stop) \equiv a(E, \{\ s\ \} + Stop);$$
while $Symbol = s$ **do begin**
$a(E, First(E) + Stop);$
$Expect(s, First(E) + \{\ s\} + Stop);$
$a(E, \{\ s\} + Stop)$
end

The procedure "*Expect*" examines the current symbol to verify that it is the expected symbol. If so, "*Expect*" obtains the next symbol from the input stream, and if not, the procedure "*SyntaxError*" is called. "*Expect*" then makes sure that the next symbol is one of the expected stop symbols, by using the procedure "*SyntaxCheck*". "*SyntaxCheck*" calls "*SyntaxError*" if the symbol is not in the stop set. The procedure "*SyntaxError*" reports that an error has occurred and discards input symbols until the current input symbol is in the synchronizing set "*Stop*".

While examples will be given when the implementation is discussed, more examples of the use of this method can be found in Brinch Hansen [Bri85], as well as in van Dijk [vDi87a]. In addition, to the use of semantic information, other exceptions to the construction method will be discussed when the implementation is discussed. Before discussing semantic analysis and code generation, the symbol table should be discussed.

7.1.2 The Symbol Table.

As was mentioned in section 4.2., a stack implemented list structured symbol table is used for the Estelle compiler. The symbol table consists of a block table, a number of object lists and a pointer to the current block in the block table. (Figure 7.1.).

Each element of the block table contains a field to store the number of temporary variables which will be on the run time stack at the present instruction being compiled, while another field holds the maximum number of temporary variables used at one time on the run time stack when executing the current block. Other fields contain a pointer to the

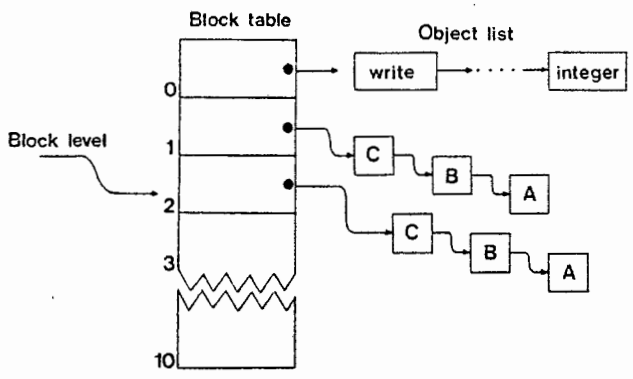

Figure 7.1: The Symbol Table.

object list associated with the current block and a boolean indicating whether the current block is a block associated with a true Estelle block or with a procedure or function block. This information is necessary as some statements cannot occur in procedure and function blocks. The rest of a table entry contains a record to store information about program labels. Labels are stored as the value of the label in the source, as well as the value associated with it in the compiler. The object lists consists of variant records which are linked together. These records can be considered the symbol table entries for various types of objects.

Symbol table entries have three fields in common, namely a field for an integer representing an identifier name, a pointer to the previous entry declared at that level (used to link the object list) and a field defining the entry type. As variant records are used to allow different attributes, this field is used as a tag deciding the attributes of the entry. Entries for the various types of object information to be stored are as follows:

(a) For array types two fields hold pointers to other entries in the table, one to the index type, and the other to element type, of the array.

(b) For module bodies a pointer is kept to the module header type associated with the body.

(c) Channel entries contain pointers to entries for both of the roles associated with the channel, as well as three pointers to lists of interactions associated with each of the roles separately, and both combined, respectively.

(d) A pointer to a value record, which is a variant record which will be discussed later, and a pointer to the constant's type, are the two fields of a constant's entry.

(e) For type synonyms an entry contains a pointer to the original type's entry.

(f) Entries for enumerated types have two fields. The one a pointer to the first identifier in the enumeration type, and the other a pointer to the last identifier.

(g) An entry for a field of a record type contains the displacement of the field from the start of the record, a pointer to the field's type, a boolean indicating whether the field is a tag field, a boolean indicating whether the field is itself in the variant part of another record type. A variant part is associated with this last boolean. When the boolean is true, another field containing a pointer to entry of the selector is visible.

(h) Interaction entries contain two fields. The one contains a pointer to the last argument in the interaction's argument list, while the other contains the total length, (in memory units), of the interaction argument list.

(i) Entries for interaction points have three fields, the first a pointer to the interaction point's type, and the other two, the block level at which was declared and the displacement from the beginning of the AR of the block.

(j) For interaction point types entries hold pointers to the entries of the channel and role associated with the interaction point type. An additional field contains an internal encoding of the queue discipline associated with the interaction point type.

(k) A module header definition's entry contains a field for the class of the module, i.e. whether it is a systemprocess, systemactivity, process or activity, and two fields for each of following: the parameter part, interaction point part and exported variable part. One field contains a pointer to the last parameter, interaction

point or exported variable, while the other contains the memory used in the case of the parameters and exported variables, and the number of interaction points defined in the case of the interaction point part.

(l) Entries for module variables contain three fields, namely a pointer to its type, the block level at which the module variable was declared, and the displacement of the module variable from the start of the AR of the block.

(m) For pointer types an entry contains a pointer to the domain type of the pointer type, i.e. the type of the object to which the pointer can point.

(n) Procedure and functions have the same type of entry with a variant part added for functions. The first field in the entry indicates the status of the declaration, i.e. whether it is has been completely defined, whether the directive external was used, whether it was defined as a forward procedure or function or whether the directive primitive was used. The next field contains a pointer to the last parameter defined in the procedure's parameterlist, while further fields contain the block level in which the procedure was defined, as well as a label indicating the start of the procedures code. A boolean field which indicates whether the entry is for a function or not is also used as the tag of a variant part. When the tag is true the entry contains the following extra fields: a pointer to the result type, a boolean indicating whether assignments to the function variable are permissible, and the displacement of the function variable in the AR of the current block.

(o) Record types have two fields in their entries, one the maximum memory used by the entire record, and the other a pointer to the entry for the last field defined for the record type.

(p) Set types have a pointer to the base type of the set in their entries.

(q) Entries for a state type contain pointers to the first and last state identifiers of the type.

(r) State sets are represented by an entry containing a pointer to a the state set constant.

(s) An entry for a subrange type contain two value pointers to the upper and lower bounds of the subrange, as well as a pointer to the type of the values which make up the subrange.

(t) Entries for the variant part of a record contain pointers to entries for the tag part associated with the variant, for the last constant in the case constant list and one for the field part of the variant. In addition, the entries also contain a field with the number of constants in the case constant list, as well as one for the amount of memory used by the variant.

(u) The entries for interaction arguments, variables, value parameters, variable parameters and exported variables are the same. They contain the block level at which the object was declared, the displacement of the object from the start of the AR of the current block, a pointer to the object's type, as well as a boolean indicating whether it is an exported variable.

(v) Entries for standard procedures and functions, and for the role of a channel contain no extra fields in the variant part. This is also true for an entry used when the attributes are not known, i.e. objects with undefined attributes.

As was mentioned in section 3.3., it is common to have some information outside the table. This is true for the above entries as the type of a variable is not stored in the variable's entry, but rather stored in another entry and linked in by a pointer. A better example is that interaction groups and constants are stored with totally different entries which are linked into the symbol table:

For the groups of interactions associated with a channel a entry containing a pointer to an interaction together with a link to a similar entry is used. A linked list of interactions can therefore be formed.

The entries used for constants all contain a pointer which can be used to link in the constant defined after the present one. Various variants are used to contain the information for specific kinds of constants. No tagfield is used in the entry as the type of variant can be derived from the symbol table pointing to the constant entry. These variants are:

(a) For character and integer constants an integer value is used to store the constant.

(b) Real constants are stored in a field of type real.

(c) Set and state set constants have a pointer to a set constant entry which is a separate entry type containing the lower and upper bounds of the set, as well as the set constant itself.

(d) The variant for a string constant contains a pointer to a string entry, a separate entry type containing the type of the string, and the string itself.

Insertions into the symbol table are performed by placing a newly created object record in the front of the linked object list associated with the current block level. Searching is performed by searching the object lists of all the existing blocks, starting with the current block and descending down the block levels until the object is found or until the last block level's object list has been unsuccessfully searched. Deletions are fortunately not necessary within blocks, making deletions very simple as current block's entries can be deleted from the table by simply decrementing the block level and discarding the pointer to the object list.

In the next sections, concerning semantic analysis and code generation, more detail will be given concerning the symbol table and its use, as well as some examples. Semantic analysis will be discussed first.

7.1.3 Semantic Analysis.

In the Estelle compiler semantic analysis is performed chiefly by the interrogation of the symbol table. In some cases semantic analysis is performed without reference to the symbol table, rather by making use of information stored in objects local to procedures which recognize syntactic structures. Whether analysis is performed with information in local variables or in the symbol tables, both scope and type analysis are performed from within procedures, and can not be separated from the syntactic analysis or code generation code. However, as scope analysis relies on the structure of the symbol table to store scope information, and type analysis makes use of the attributes which are contained in the symbol table, each will be discussed separately.

7.1.3.1. Scope Analysis.

A specification of Estelle's scope rules can be found in [ISO87]. The specification is written in language which precludes ambiguity, but sadly, does not contribute to making the specification clear or easily understandable. To simplify matters, a number of simple rules, which

reflect the essence of the scope rules, as well as some concepts needed, will be given.

Estelle, in common with Pascal, Ada and C, is one of the many languages that uses the so called **lexical- or static- scope** rules, which allow scope to be determined by the analysis of the specification text alone.

Estelle specifications combine related definitions and statements into syntactic units called blocks. There are 5 kinds of blocks, namely

(a) The standard block.

(b) The specification block.

(c) Procedure and function blocks.

(d) Module blocks.

(e) Transition blocks.

The standard block contains the whole specification, while the specification block contains the definitions and statements local to it and procedure-, function-, module- and transition blocks. These in turn may contain other blocks.

The purpose of the block concept is to confine the use of an object to the block in which it was defined, apart from explicit exportation, which allows access from beyond the block. The simple scope rules now follow:

(a) All objects, i.e. constants, types, variables, procedures, functions, channels, modules, interaction points etc, defined in the same block must have different names.

(b) All data objects excluding procedures and functions defined in a block are normally known from the end of their definition to the end of the block. A procedure or function defined in a block B is normally known from the beginning of the procedure block to the end of the block B. This allows recursive procedures and functions.

(c) Consider a block Q that defines an object x. If Q contains a block R that defines another object named x, the first object is unknown in the scope of the second object.

The method used in the Estelle Compiler to keep track of scope information is as follows: the object list associated with a block level can be considered to be a stack of definitions. When a new object is defined, the object is inserted into the symbol table, or conceptually, the object is pushed onto the stack. Before compilation starts, all the standard procedures, functions and types are pushed onto the stack. At the end of a block, the compiler removes all the identifiers defined in that block from the stack. When an occurrence of an object is encountered, the scope analyzer searches the stack, starting at the top, to find the object. If it was not found, an error is generated, and the object is then defined as an object of the class *"Undefined"*, so that further occurrences of the object will not cause further error messages.

While the design of the symbol table simplifies scope analysis, as the structure defines the scope of objects in blocks, the Estelle scope rules define exceptions to block based scope rules. These exceptions will be discussed when the implementation is discussed.

Type analysis will be discussed next.

7.1.3.2. Type Analysis.

The procedures which are used during scope analysis to manipulate the symbol table, i.e. to enter new objects, are also used during type analysis. The procedure which places a new record into the symbol table, is also used to place the class of an object into an object record, thus allowing other attributes to be entered in the object record.

For example, in section 7.2. the attributes associated with each type of entry were discussed. Recall that for constants two attributes are stored in the variant part of the symbol table entry, namely a pointer to the constant's value record, and a pointer to its type. To store the integer constant *"MaxLevel"* in the constant definition
"MaxLevel = 10;", the object record will contain the identifier *"MaxLevel"*, (actually an identifier index), the class *"Constant"*, a pointer to a value record containing the constant value 10, and a pointer to the object record for the standard type *"integer"* in the fields for the identifier, class, value pointer, and type pointer respectively.

Type analysis can be performed by the comparison of two object records' information. For example, to check that the types of two iden-

tifiers are the same, the pointers to the two types can be compared. If the pointers are not the same, the types are not equal. However, for type equivalence, type compatibility and assignment compatibility the attributes in the object records have to be examined. As each derived type contains pointers to the type from which it was derived, type checks can be performed by travelling down the links formed by the type pointers of objects until equivalence can be proven or disproved. By providing a number of standard pointers to dummy object records, the standard types of Estelle can be incorporated, thus creating a type system.

In the Estelle compiler a number of procedures can be called to perform type equality, type equivalence, type compatibility and assignment compatibility checks. A type called "*Typeuniversal*", which is used in the same way as the class "*Undefined*", allows the suppression of extra error messages, as it is defined to be the same as any other type.

An example of how a type check is performed is for the object "*MaxLevel*" which was defined before. Suppose that a type rule requires that an object is an integer constant. To test the rule on "*MaxLevel*", the class of "*MaxLevel*" is compared with the class "*Constant*". As "*MaxLevel*" 's class indicates that it is a constant, it remains to be tested that the constant is an integer. Because "*MaxLevel*" is a constant, its object record contains a type attribute, i.e. a pointer to the constant's type. By comparing this type pointer with the pointer "*TypeInteger*", "*MaxLevel*" will be shown to be an integer constant, as the two pointers are identical.

Code generation, which is entwined in the procedures of the recursive-descent compiler, will be discussed next.

7.1.4 Code Generation.

The final task performed by Pass Two is the generation of code. The code in question is pseudo-code for a hypothetical machine, the **Estelle Machine**. Naturally, it is necessary to know the instruction set of the Estelle Machine, before code can be generated. Because the Estelle Machine was specifically designed to execute Estelle specifications, there is a close association between Estelle and the pseudo code which is generated to implement each construct. Code generation can therefore be discussed without reference to the function of specific

Estelle Code instructions. The Estelle Machine will be discussed in chapter 9.

The design of the code generator was govern by 3 rules, due to Brinch Hansen [Bri85], namely:

(a) For every syntax rule in Estelle, a code rule must be written that defines the corresponding sequences of instructions to be generated.

(b) Every instruction should preferably have the same name as the Estelle symbol it represents.

(c) A code rule should preferably have the same syntactic structure as the corresponding syntax rule in Estelle.

There are a number of classes of code to be generated. The general scheme for a class will be discussed as well as an example. In the case of Estelle constructs, full details will be given.

7.1.4.1. Object Addressing.

The addressing of objects, i.e. interaction points, module variables, interaction arguments, variables and parameters, by relative level numbers and displacements is based on the simple dynamic memory management scheme discussed in section 3.4. All objects are addressed with a relative level number which defines the number of levels to travel up the static link formed by the static pointers of the activation records (AR's) on the run time stack to arrive at the AR in which the object can be found. The displacement defines the displacement of the object from the start of the AR's local data area. The addresses defined are used in instructions sent to the code file. In a similar way, addresses are defined for indexed variables, indexed interaction points and field variables by adding a displacement from the start of an object.

7.1.4.2. Expression Code.

The procedures of the parser which recognize the BNF rules associated with expressions, ("*Expression*", "*Term*", "*Factor*" etc), are used to generate code for the expressions. The code generated for expressions

always ensures that the result of an expression remains on the top of the stack. Note that the grammar rules used to describe expressions are designed in such a way as to force precedence of operators by the order in which parts of an expression are recognized, and therefore the order in which code is generated for each part.

An interesting feature of the code generated for expressions is the use of a parameter for the code generated for mathematical expressions. For example, the code for the multiplication operator in the expression "$a * b$" is "*Multiply* 0" for "a" and "b" both integers and for "a" and "b" both reals "*Multiply* 1". Note that these instructions are prime candidates for peephole optimization by replacement by special purpose instructions.

7.1.4.3. Statement Code.

For assignment statements, code is generated to place the address of a variable on the stack, and at the end of an assignment statement, code is generated to assign the value on top of the stack, which will be the result of expression evaluation, to that address.

For while-statements, for example, "**while** B **do** S", apart from the code generated to evaluate B and the code generated for the statement(s) S, code must be generated for the control flow. In fact two jumps are generated:

L_1: (* Evaluation code for B *)
if not B **then Jump**(L_2)
(* Code for S *)
Jump(L_1)
L_2:

Note that instructions are generated for the jumps, the first a conditional jump, and the second unconditional. The two labels L_1 and L_2 do not have addresses when the instructions are generated. Two pseudo-instructions (i.e. directives) are generated. These will be used by Pass Three to resolve the addresses. Pass Two therefore generates the instructions "**DefAddr**(L_1)" and "**DefAddr**(L_2)" at the positions of the two labels. Labels are each given a unique numerical value which is sent to the code file.

Code is generated for if-statements in a similar way, with "**DefAddr**" directives used to define labels used in the jumps generated. The other statements which also occur in the Pascal subset of Estelle, for example, repeat and for- statements, will not be discussed further, as the code generated is similar to that for existing Pascal compilers.

7.1.4.4. Procedure Code.

During code generation for procedure-statements, another directive is used to solve problems caused by the fact that the storage needed for local and temporary variables, as well as the address of the procedure's statement part is not known when the procedure heading is parsed, and code generated for it. "**DefArg**" is used to define the value of a label that was not previously known. Thus for a procedure- statement, say,

procedure P;
var
  i, j: *integer*;
begin
  (* Statement List SL *)
end;

The following code would be generated:

DefAddr(*ProcLabel*)
(* Procedure heading instruction *)
DefAddr(*BeginLabel*)
(* Instructions for SL *)
DefArg(*VarLabel*, *VarLength*)
DefArg(*TempLabel*, *MaxTemp*)
(* End of procedure instruction *)

After the instructions for SL have been generated, it is known how many temporary values will be needed on the stack, and how much memory is needed for variables. Note that this cannot be done straight after the variable definition part of the procedure block as some procedure definitions may have been nested in the procedure.

For procedure calls, instructions are generated for the actual parameters, followed by instructions for the procedure call itself. Function calls are very similar, differing only in that the result is considered to be an extra parameter.

Code generation of individual constructs of Estelle will now be discussed in detail. The code generated was designed to execute in a run time environment based on preliminary notes on the subject by Dreyer ([Dre87], [Dre87a]).

7.1.4.5. Estelle Primitives.

A number of Estelle statements can be considered primitives as they can be directly mapped to a single E-code machine instruction. The instructions needed to place values on the stack used by the E-code instruction mapped from the statement are not considered, as they where discussed in sections 7.1.4.1. and 7.1.4.2.

For init-statements the following code is generated: the pseudo-code instruction "**Init**" followed by the following arguments: a module type number, size of the static local data area, number of temporary stack variables used, address of the initialization code, address of the executable code, size of the parameter area, size of the external data area, number of external interaction points and the line number in the source text of the init-statement. For release- statements the pseudo code instruction "**Release**" is generated followed by the line number of the statement.

For connect- and disconnect-statement instructions the pseudo-code instructions "**Connect**" and "**Disconnect**" are generated respectively, as well as arguments indicating whether the interaction points are internal or external, while for attach- and detach-statements the instructions "**Attach**" and "**Detach**" are generated. In addition, each instruction is followed by the source line number.

For output-statements the instruction "**Output**" is generated with the length of the interaction argument list and the source text line number as arguments.

7.1.4.6. Exist, All and Forone Code.

For an exist-expression there are two forms, namely the domain-list form and the module-domain form. The code generated for the example "**exist** x: T **suchthat** B" (where "B" is a boolean expression) is as follows for the module-domain form:

```
   Scan(ModuleType, LineNo)
L1:(* Code for Address of x *)
   NextInstance(L2, LineNo)
   (* Evaluation code for B *)
   Do(L1)
   Constant(1)
L2:
```

Where "*ModuleType*" is an integer uniquely identifying a module header. Note that the labels in the code will be defined by using the directives "**DefAddr**" and "**DefArg**".

For the domain-list form the code generated is the equivalent of the code generated for the statements: Say "T" is the ordinal type "0..10" and let the top of the variable stack with the result of the exist expression be called "$St[sr]$".

```
x := 0;
St[sr] := false;
while (x <= 10) and (not St[sr]) do
   if B then St[sr] := true
   else x := x + 1;
```

For an all-statement, say "**all** x: T **do** S" the code generated for the module-domain form is

```
   Scan(ModuleType, LineNo)
L1:(* Code for Address of x *)
   NextInstance(L2, LineNo)
   (* Code for S *)
   Jump(L1)
L2:Pop(1)
```

The domain-list form is the equivalent of the statement
"**for** $x := 0$ **to** 10 **do** S" (With the variable "x" being declared for the for-statement).

The code generated for a forone-statement is a combination of an if-statement and an exist-expression, for example,
"**forone** x: T **suchthat** B **do** S_1 **otherwise** S_2;" and
"**if exist** x: T **suchthat** B **then** S_1 **else** S_2;"
are equivalent statements.

Note that when an identifier-list occurs in the domain-list form, all identifiers are mapped to the same address in memory. This means that the entire vector can incremented by one incrementation.

7.1.4.7. Transition Code.

A transition declaration is very similar to a procedure definition as it represents the definition of code which is not directly executed. The code generated must therefore make provision for returning to the code which started execution the transition. The code for a transition can be seen as the following:

L_1: (* Code for the when- or delay-clause. *)
L_2: (* Code for the provided-clause. *)
(* Code for the procedure and function declarations. *)
L_3: (* Code for the statement-part. *)
(* Code to remove interaction in when-clause *)
(* from queue in the case of an input transition. *)

Note that L_1 and L_2 can be in any order. The code generated for the when-clause:
"**when** $ip.interaction(ia_1, ia_2, \ldots, ia_n)$" is as follows:

L_1: **Variable**(l, d)
Ip($LineNo$)
Interaction($InteractionIdentification$, $LineNo$)
Return(1)

Thus, the result on the top of the stack is the boolean indicating whether the first interaction in the queue associated with the connection endpoint at the ip is the interaction in the when-clause.

For a delay-clause "**delay**(e_1, e_2)" the following code is generated: (Where "e_1" and "e_2" are integer expressions.)

L_1: (* Code for e_1 *)
(* Code for $e2$ *)
Return(2)

For the two cases "**delay**(e_1)", and "**delay**(e_1, *)" the code generated is respectively:

L_1: (* Code for e_1 *)
Copy
Return(2), and

L_1: (* Code for e_1 *)
Constant(−1)
Return(2)

A provided-clause, say "**provided** B", where "B" is a boolean expression, while be compiled as follows:

L_2: (* Code for B *)
Return(1)

Note that a "**provided otherwise**" is not recognized by the Estelle compiler. This is due to the fact that the short- hand notation is not supported. Any-clauses are thus also not supported.

The following instruction is then generated for the transition declaration:
Transition(*From*, *To*, *Priority*, *Input*, L_1, L_2, L_3, *LineNo*),
where "*From*" is a state set constant indicating from which state(s) the transition may be made, "*To*" is the state to which the current state must change, (−1 indicates the same state), "*Priority*" contains the priority of the transition, while "*Input*" is a boolean indicating whether the transition is an input transition or a spontaneous transition, "L_1" contains the address of the when-clause or delay-clause for input and spontaneous transitions respectively, and "L_2" and "L_3" contain the addresses of the provided-clause and the statement code respectively.

The initialization part of a module is a form of the above transition and will not be discussed further.

7.1.4.8. Module and Specification Code.

The code generated for modules and the specification is very similar. Both a module and specification definition contain a body definition which has three parts, namely the declaration-part, initialization-part and transition- declaration-part.

Code for the body definition of a module or specification is the following:

L_1: (* Code for Declaration-part *)
(* Code for Initialization-part *)
L_2: (* Code for Transition-declaration-part *)

The code generated for the transition-declaration-part, and therefore the initialization-part, has been discussed. The code generated for the declaration-part consists of instructions to define the internal interaction points of the module or specification body. For example, the code generated for the interaction point definition
"ip N: $Network-access-point(User)$ **common queue**;"
is as follows:
DefineIp($IpLevel$, $IpDispl$, 0, $LineNo$)

"$IpLevel$" and "$IpDispl$" are the relative level and displacement of the ip in the module/specification block, while the third parameter indicates the queuing discipline "**common queue**". It is necessary to define the ip in this way as the address of the queue associated with the ip is placed at the address "$IpLevel$, $IpDispl$".

In addition to the above code generated in the body definition part of the specification, the instruction
Specification($Vars$, $Temps$, L_1, L_2, $LineNo$)
is generated for specifications at the start of the code. Where "$Vars$" is a label used to indicate the size of the internal data space, "$Temps$" is a label used in the same way for temporary variables on the stack and L_1 and L_2 are labels for the two parts as above. The equivalent instruction for modules is the "**Init**" instruction discussed in section 7.1.4.5.

7.2 The Implementation.

Before discussing the main module, called "*Syntax Analysis*", the various service modules used by Pass Two will be discussed:

7.2.1 Service Modules.

The syntactic, scope and type analysis, as well as the code generation routines of Pass Two use various services presented in the service modules "*Administration*", "*Input*", "*Output*", "*Variable Addressing*", "*Labels*" and "*Temporaries*". In addition, while the modules "*Scope Analysis*" and "*Type Analysis*" form an integral component of Pass Two, they present services which are called by the driver of Pass Two, the module "*SyntaxAnalysis*". These modules will therefore be considered to be service modules for the purposes of this implementation description.

7.2.1.1. "*Administration*".

"*Administration*" presents the services "*Emit*", "*EmitReal*", "*NewLine*", "*StartMessage*" and "*FinalMessage*". "*Emit*" provides the service of writing an integer value to the intermediate code file, while "*EmitReal*" provides the service of writing a real value to the file. Note that both "*Emit*" and "*EmitReal*" test a boolean "*Emitting*", which is made false by the error reporting procedure. While "*Emitting*" is true, output is sent to the intermediate code file. When an error has occurred, "*Emitting*" is false, and "*Emit*" and "*EmitReal*" suspend output. "*NewLine*" updates the current line number used for error reporting, as well as setting a boolean, "*CorrectLine*", which is used to suppress error messages for line numbers that have already occurred in error messages. "*StartMessage*" and "*FinalMessage*" allow messages to printed when execution starts and ends, providing feedback to the user.

7.2.1.2. "*Errors*".

The services "*Error*", "*Warning*" and "*InRange*" are provided by the service module "*Errors*". "*Error*" is used to print error mes-

sages for the rest of the program. An enumeration type "*ErrorKind*" contains defines an internal representation for errors which can occur during compilation. The possible error messages are presented in Appendix C. Note that two internal compiler errors are also reported by "*Error*", namely when the number of permissible labels used in the code becomes to big, and when the number of levels permissible in the block table is exceeded. These errors cause abortion of the compilation. "*Error*" prints out an error code, i.e. from where the procedure was called, the line number of the source where the error occurred and the error message as indicated by a parameter of type "*ErrorKind*". The booleans "*Emitting*" and "*CorrectLine*", which have been discussed are also set by "*Error*".

"*Warning*" provides a similar service to "*Error*", except the enumeration type is "*WarningKind*", which enumerates the warnings given by the compiler. These warning messages, which can also be found in Appendix C, concern limitations of the implementation, as opposed to errors. For example, when a "new(p)" standard procedure is used in the form "new(p, t_1, t_2, t_3)", a warning is given as tagged "new" 's are not implemented. A list of the restrictions placed on specifications is given in Appendix D.

The service "*InRange*" is used to test that a value is within a range. If not an error message is generated via a call to "*Error*".

7.2.1.3. "*Input*".

"*Input*" provides the service "*NextSymbol*", which gets the next token from the intermediate file produced by Pass1. In the case of tokens which have an argument, the argument is also read.

7.2.1.4. "*Output*".

"*Output*" provides the services "*Emit1*", "*Emit2*", "*Emit3*", "*Emit4*", "*Emit5*" and "*Emit6*", which send the ordinal value of an operator of the pseudo code's representation to the intermediate code file, via "*Emit*", along with 0, 1, 2, 3, 4 and 5 integer arguments respectively. In addition, the service "*EmitSet*", which sends the internal representation of a set to the intermediate code file as a series of "*Constant2*"/integer pairs.

7.2.1.5. "*ScopeAnalysis*".

"*ScopeAnalysis*" contains the services used to perform scope analysis in the Estelle compiler, namely "*Search*", "*Define*", "*FindDefinition*", "*Find*", "*NewBlock*" and "*EndBlock*". The module can therefore be considered to be the symbol table manager. Before discussing the various services, the implementation of the symbol table will be discussed.

The compiler keeps the current block level in a variable called "*BlockLevel*". The data structure to store this information is declared as:

```
const
   MaxLevel = 10;
type
   Class = (ArrayType, Body, Constantx, ChannelType,
            DerivedType, EnumeratedType, Field, Ip, IpType,
            Interaction, InteractionArg, ModuleType, ModVar,
            PointerType, Procedur, RecordType, RoleType,
            SetType, StandardFunc, StandardProc,
            StandardType, Statex, StateSetx, SubrangeType,
            Undefined, VariantType, Variable, ValueParameter,
            VarParameter);
   Pointer = ↑ ObjectRecord;
   ObjectRecord =
      record
         Identifier: integer;
         Previous: Pointer;
         case Kind: Class of
            ArrayType: (...);
            Body: (...);
                 ⋮
            VarParameter: (...);
      end;
   LabelRecord =
      record
         Valuex, Number: integer
      end;
   BlockRecord =
```

```
    record
      TempLength, MaxTemp: integer;
      LastObject: Pointer;
      Labelx: LabelRecord;
      EstelleBlock: boolean
    end;
  BlockTable = array [0..MaxLevel] of BlockRecord;
```

Where "*Class*" is the enumerated type defining the variants of object records used in the symbol table. The definition of each variant will be given in section 7.2.1.6. when the module "*TypeAnalysis*" is discussed. "*LabelRecord*" defines the record used to store information about labels, while "*BlockRecord*" defines the entries in the stack "*Blockx*" of type "*BlockTable*". These data types define an implementation of the symbol table described in section 7.1.2.

The procedure "*Search*", searches the object list associated with a block record to find out whether an object was defined in that level, while "*Define*", is used to define a new object. If it has already been defined, an error message is reported. The service "*FindDefinition*", which makes use of "*Search*", is used to find out whether an identifier has been defined. The stack "*BlockTable*" is searched from the current block down to block level 0. If an object is found, "*FindDefinition*" returns a pointer to the object record. The service "*Find*" uses "*FindDefinition*" to find an identifier's object record. If "*FindDefinition*" indicates that no object record occurs for that identifier, an error is reported and the identifier is defined with the class "*Undefined*".

Note that the distinction between the two procedures, namely that one announces an error and defines the identifier, while the other simply indicates the non-existence of an object record, is important. An example illustrating the difference will be discussed in section 7.2.4.4.

"*NewBlock*" is used to increment the block level and create a new (empty) list of objects for the new block. "*NewBlock*" also initializes the information in the new block record, i.e. the number of temporaries, the maximum number of temporaries, as well as the two fields associated with labels, are initialized with the value zero, while the field boolean "*EstelleBlock*" is given the value of the parameter with which "*NewBlock*" was called. At the end of a block the procedure

"*EndBlock*" is used to decrement the block level and discard the list of objects.

The service "*Redefine*" is used to redefine a symbol table entry. Copying the entry itself is insufficient as chaos results. The links of the original entry are used at the current level and the symbol table becomes disorganized. "*Redefine*" restores the new links after copying the entry itself, solving this problem. Another service, "*StandardBlock*", is used to initialize the symbol table.

7.2.1.6. "*TypeAnalysis*".

The service module "*TypeAnalysis*" contains the services "*IsIn*", "*SameStringType*", "*CheckTypes*", "*EqualTypes*", "*CompatibleTypes*", "*AssignmentCompatible*", "*TypeError*", "*KindError*" and "*CheckList*" which are used to perform type analysis in the Estelle compiler,

In section 7.1.2. the variants used for the different kinds of object records was described. As the attributes stored in the object records' variant parts are accessed during type analysis, the variants will be listed. A number of data types used in the definition of the variants are needed first:

```
type
  GroupPointer = ↑ GroupRecord;
  GroupRecord =
    record
      Interaction: Pointer; Next: GroupPointer
    end;
```

The "*GroupPointer*" and "*GroupRecord*" types are used together to allow a list of interactions, used in the definition of channel types, to be formed.

```
const
  SetSize = 4;
type
  SetConstantType = packed array[0..SetSize − 1] of word;
```

"*SetConstantType*" defines the internal representation of sets in the Estelle compiler. Note that the type "*word*" is a feature of the implementation language and defines a 16-bit word. In this type it is used as a substitute for a packed array of bit, which is not implemented by *MicroSoft Pascal*. As the set size is defined to be four 16-bit words, sets may contain a maximum of 64 elements. However, by changing the constant "*SetSize*" the restriction can be changed.

```
type
   SetConstantPointer = ↑ SetConstantRecord;
   SetConstantRecord =
      record
         SetLowerBound, SetUpperBound: integer;
         SetValuex: SetConstantType
      end;
```

"*SetConstantPointer*" and "*SetConstantRecord*" together define dynamic records used to store information about set constants.

```
type
   StringPointer = ↑ StringRecord;
   StringRecord =
      record
         DefinedType: Pointer; Valuex: StringType
      end;
```

"*StringPointer*" and "*StringRecord*" are used to define dynamic records used to store information about string constants. The type "*StringType*" is defined to be a variable string type of maximum length 80 characters. The variable string type is an implementation feature of *MicroSoftPascal*.

```
type
   ConstantKind = (IntegerKind, RealKind, CharKind, SetKind,
                   StringKind, StateSetKind);
   ValuePointer = ↑ ValueRecord;
   ValueRecord =
      record
         Next: Pointer;
```

```
        case ConstantKind of
            CharKind,
            IntegerKind: (IntegerValue: integer);
            RealKind: (RealValue: real);
            SetKind,
            StateSetKind: (SetValue: SetConstantPointer);
            StringKind: (StringValue: StringPointer);
    end;
```

"*ValuePointer*" and "*ValueRecord*" define dynamic records to store various types of constants. Note that the type "*ConstantKind*" is used for tagless variants.

```
type
    DeclarationStatus = (Complete, Externalx, Partial, Primitive);
```

The above type, "*DeclarationStatus*", is used to distinguish between stages of procedure or function definition. This declaration status is explained in section 7.1.2.

The different variants follow: As the enumeration is the same as in section 7.1.2. and the nomenclature leaves no doubt to the contents of the fields, the variants will not be discussed further.

```
(a) ArrayType : (IndexType, ElementType: Pointer);
(b) Body : (Header: Pointer);
(c) ChannelType : (User, Provider: Pointer;
                   UserInteractions: GroupPointer;
                   ProviderInteractions: GroupPointer;
                   BothInteractions: GroupPointer);
(d) Constantx : (ConstValue: ValuePointer;
                 ConstType: Pointer);
(e) DerivedType : (DeclaredType: Pointer);
(f) EnumeratedType : (First, Last: Pointer);
(g) Field : (FieldDispl: integer;
             FieldType: Pointer;
             TagField: boolean;
             case Tagged: boolean of
                 true: (Selector: Pointer);
                 false: ());
```

(h) *Interaction* : (*LastArg*: *Pointer*;
ArgLength: *integer*);
(i) *Ip* : (*IpTypex*: *Pointer*;
IpLevel, *IpDispl*: *integer*);
(j) *IpType* : (*Channel*, *Role*: *Pointer*;
Discipline: *integer*);
(k) *ModuleType* : (*ProcessClass*: *integer*;
LastParameter: *Pointer*;
ParameterLength: *integer*;
LastIp: *Pointer*; *IpNumber*: *integer*;
LastExported: *Pointer*;
ExportedLength: *integer*);
(l) *ModVar* : (*ModVarType*: *Pointer*;
ModVarLevel, *ModVarDispl*: *integer*);
(m) *PointerType* : (*DomainType*: *Pointer*);
(n) *Procedur* : (*Status*: *DeclarationStatus*;
LastParam: *Pointer*;
ProcLevel, *ProcLabel*: *integer*;
case *IsFunction*: *boolean* of
true: (*ResultType*: *Pointer*;
CanAssign: *boolean*;
ValDispl: *integer*);
false: ());
(o) *RecordType* : (*RecordLength*: *integer*;
LastField: *Pointer*);
(p) *SetType* : (*BaseType*: *Pointer*);
(q) *Statex* : (*FirstState*, *LastState*: *Pointer*);
(r) *StateSetx* : (*StateSetValuex*: *Pointer*);
(s) *SubrangeType* : (*LowerBound*: *ValuePointer*;
UpperBound: *ValuePointer*;
RangeType: *Pointer*);
(t) *VariantType* : (*TagPart*, *LastConstant*: *Pointer*;
FieldPart: *Pointer*;
Constants, *Length*: *integer*);
(u) *InteractionArg*,
Variable,
ValueParameter,
VarParameter : (*VarLevel*, *VarDispl*: *integer*;
VarType: *Pointer*;

Exported: *boolean*);
(v) *RoleType* : ();
StandardFunc : ();
StandardProc : ();
StandardType : ();
Undefined : ();

The service "*IsIn*" is used to ascertain whether an object is of a certain type or kind. For example, the function call "*IsIn*(*Object*, *Pointers*)" will return the boolean value true if the object is the definition of a pointer type. The possible types and kinds that can be checked using this service are defined by the enumerated type "*TypeClass*":

```
type
  TypeClass = (Integers, Reals, Chars, Booleans,
               Enumerables, Enumerateds, Numericals,
               Ordinals, Pointers, Ranges, Sets,
               SimpleTypes, Strings);
```

For "*Integers*", "*Reals*", "*Characters*" and "*Booleans*", "*IsIn*" consists of a comparison between the first parameter and the pointer to the dummy object record for the standard type, i.e. the pointers "*TypeInteger*", "*TypeReal*", "*TypeChar*" and "*TypeBoolean*", respectively. For "*Numericals*" "*IsIn*" calls itself recursively to ascertain whether the type is the integer or real standard type. "*IsIn*" compares the "*Kind*" of the object associated with the first parameter with the classes "*EnumeratedType*", "*PointerType*", "*SubrangeType*", and "*SetType*" for "*Enumerateds*", "*Pointers*", "*Ranges*" and "*Sets*", respectively. For the rest of the type kinds, recursive calls with both types of comparisons are used.

The service "*SameStringType*" is used to test whether two type pointers are associated with the same or an equivalent string type. For example, the string constant 'aaaa' is considered to be of the same string type as the type "*StrType* = array[1..4] of *char*".

"*CheckTypes*" and "*EqualTypes*" perform the same service, namely to test whether two types are identical or not. Both call an error routine if the types differ and neither of the types is the standard type "*TypeUniversal*", which is used to suppress spurious error messages.

They differ in that "*EqualTypes*" is a function returning a boolean indicating the result of the comparison.

"*CompatibleTypes*" and "*AssignmentCompatible*" are very similar in purpose. While "*CompatibleTypes*" tests the compatibility of two types, "*AssignmentCompatible*" tests whether a variable of the second type can be assigned to a variable of the first type. Both return integer results indicating not only whether the types are compatible, but also what compatibility rule applied. For example, for the variables a and b, where a is an integer and b a real, "$CompatibleTypes(type_a, type_b)$" returns the compatibility value 7. Thus, by testing the compatibility of two variables, the types of both can also be ascertained. This is better than finding out that the types are compatible, and then having to find what the types are. "*AssignmentCompatible*" makes use of "*CompatibleTypes*", as the result of the compatibility test differs only from "*AssignmentCompatible*" in the case used as an example.

"*TypeError*" and "*KindError*" are used in the reporting of semantic errors. Because the procedures test to see whether the kind of the object is "*Undefined*", or the type is "*TypeUniversal*", for "*KindError*" and "*TypeError*", respectively, spurious error messages are avoided.

For "*CheckList*", which is used to test whether repetitions occur in a case constant list, the following type is needed:

```
type
   ListPointer = ↑ ListRecord;
   ListRecord =
      record
         Next: ListPointer;
         LastConstant: Pointer;
         Number: integer
      end;
```

The scope analysis procedures are also used during type analysis. "*Define*" places the class of an object into an object record, while "*Find*" creates a record of "*Undefined*" class if an object was not found.

7.2.1.7. "*VariableAddressing*".

The module "*VariableAddressing*" provides services used during code generation, namely "*TypeLength*", "*VariableAddressing*", "*ParameterAddressing*", and "*FieldAddressing*". "*TypeLength*" is used to ascertain the storage requirements of an object, while "*VariableAddressing*" and "*ParameterAddressing*" are used to assign variables a displacement from the start of a block. "*FieldAddressing*" is used to assign fields of a record displacements from the start of the record variable.

7.2.1.8. "*Labels*".

"*Labels*" provides two services concerned with two types of labels, namely labels associated with the pseudo code generated, and labels associated with the goto-statement in Estelle. The service "*NewLabel*" is used to generate a new label used for code generation, while "*DefineLabel*" is used to associate a label defined by "*NewLabel*" with a label from Estelle. "*DefineLabel*" places this information in the label record associated with the current block.

7.2.1.9. "*Temporaries.*"

The module "*Temporaries*" provides two services used during the generation of code, namely "*Push*" and "*Pop*". "*Push*" is used to define the address of temporary storage on the stack which can be used in expression evaluation for temporary results. "*Pop*" is used to define the reclaiming of temporary stack storage after expression evaluation.

7.2.1.10. "*Initialization.*"

The module "*Initialization*" contains one procedure, "*Initialize*", which is used to initialize the sets used for error recovery during syntax analysis.

The rest of Pass Two consists of the module "*Syntax Analysis*", which will be discussed next.

7.2.2 "*Syntax Analysis*".

The function "*Estelle*" is used to find out whether the present block is an Estelle block or not. (I.e. whether it is a procedure or function block.) The procedures "*Expect*", "*SyntaxError*" and "*SyntaxCheck*" have been discussed. The procedure "*ExpectIdentifier*" is similar to "*Expect*", as it is used by other procedures to parse a single symbol, namely the identifier symbol. It is a separate procedure as the procedure returns a parameter indicating the value of the identifier.

The rest of the module consists of procedures to perform syntactic and semantic analysis as well as code generation of constructs of Estelle. The source text is documented with the grammar rule before each procedure and the procedures of the module will not be discussed further, except in cases where interesting features can be illuminated. However, the main program of Pass Two will be discussed first.

The main program consists of a number of procedure calls, namely "*Administration*", "*Initialize*", "*Errors*", "*NextSymbol*", "*StandardBlock*", "*Specification*" and "*FinalMessage*". "*Administration*" initializes the service module "*Administration*" and uses "*StartMessage*" to print detail of the compiler, while "*Errors*" initializes the error service module, and "*Initialize*" initializes the sets and variables during syntactic analysis. "*NextSymbol*" is used to read the first symbol from the source file, therefore placing the parser in a well-defined state ready for parsing. "*StandardBlock*" sets up the initial block record associated with the standard block and initializes the symbol table. "*Specification*" then is the start of the recursive-descent parsing of the specification, calling the other procedures in this module to perform syntactic and semantic analysis as well as code generation.

7.2.3 Examples.

The constructs of a language can be divided into two classes, namely definition constructs and executive constructs. An example of both will be given, representing insertions into the symbol table, i.e. information gathering, and symbol table interrogation and code generation respectively. The following is the code from the compiler used for a

constant definition (the lines are numbered for easy reference):

```
"ConstantDefinition".

{ ConstantDefinition = ConstantIdentifier "=" }
{                      ( Constant | "any" TypeIdentifier) }

1  procedure ConstantDefinition(Stop: Symbols);
2  var
3     Identifier: integer; Valuex: ValuePointer;
4     Constx, Typex: Pointer; Stop2 : Symbols;
5  begin
6     Stop2 := Stop + ConstantSymbols + [Any1];
7     ExpectIdentifier(Identifier, [Equal1] + Stop2);
8     Expect(Equal1, Stop2);
9     if Symbol = Any1 then begin
10       Warning(210, AnyConst3);
11       Expect(Any1, [Identifier1] + Stop);
12       TypeIdentifier(Typex, Stop);
13       New(Valuex);
14       Valuex ↑.IntegerValue := 0
15    end (* if *)
16    else Constant(Valuex, Typex, Stop);
17    Define(Identifier, Constantx, Constx);
18    Constx ↑.ConstValue := Valuex;
19    Constx ↑.ConstType := Typex
20 end; (* ConstantDefinition *)
```

In line 7 the syntactic analysis commences. The first symbol expected is an identifier followed by "=", "any", all the symbols possible for a constant, and all the symbols which can follow a constant definition. The parameter "*Identifier*" returns the value associated with an identifier.

In line 8 the symbol "=" is expected with the follow set of the previous statement reduced by the symbol "=".

In line 9 a choice is made between two possible syntactic alternates, if the current symbol is the symbol "any", then the alternate < "any"

TypeIdentifier > is pursued in the if-statement, otherwise the else-part, i.e. line 16, is used to recognize the alternate < Constant >.

In line 10 a warning message is generated as the use of the incomplete form of a constant definition means that code can not be generated fully, as information is lacking. Code generation is therefore halted and only analysis is performed.

In line 11 the symbol "any" is expected and can only be followed by an identifier and all the symbols associated with constructs which occur after a constant definitions.

In line 12 an identifier is expected. The procedure "*TypeIdentifier*" is a special form of "*ExpectIdentifier*" in which only identifiers associated with types are permissible. If the next symbol was an identifier, and the class of the identifier was some kind of type, the parameter "*Typex*" returns a pointer to the symbol table entry for the type. Otherwise, an error is reported and "*Typex*" contains a pointer to a dummy symbol table entry of the class "*Undefined*".

In line 13 a new value record is created and associated with the pointer "*Valuex*" and in line 14 the value record is given the integer value 0.

Line 15 ends the if-statement.

As was explained, line 16 is the code associated with the alternate construct < Constant >. The procedure "*Constant*" is called with two parameters, namely a pointer to a value record with the constant's value and a pointer to the constant's type.

In line 17 the identifier expected in line 8 is defined in the symbol table with the class "*Constantx*". The pointer "*Constx*" contains the pointer to the symbol table entry. Note that at this level of the compiler, errors incurred by ambiguous identifiers are ignored, as they are handled by "*Define*" and similar procedures. (In line 16 any error is handled by "*Constant*" itself, and does not effect "*ConstantDefinition*" at all.)

In line 18 and 19 the symbol table entry's attributes are entered, namely the value and the type, while line 20 concludes the procedure "*ConstantDefinition*".

The symbol table now contains information concerning the constant defined in this constant definition. An example of a procedure which generates code for a construct is "*WhileStatement*".

"*WhileStatement*".

{ *WhileStatement* = "while" *BooleanExpression* "do" *Statement* }

```
1  procedure WhileStatement(Stop: Symbols);
2  var
3      Label1, Label2: integer; ExprType: Pointer;
4  begin
5      NewLabel(Label1);
6      Emit2(DefAddr2, Label1);
7      Expect(While1, ExprSyms + [Do1] + StatementSymbols + Stop);
8      Expression(ExprType, [Do1] + StatementSymbols + Stop);
9      CheckTypes(ExprType, TypeBoolean);
10     Expect(Do1, StatementSymbols + Stop);
11     NewLabel(Label2);
12     Emit2(Do2, Label2);
13     Pop(1);
14     Statement(Stop);
15     Emit2(Goto2, Label1);
16     Emit2(DefAddr2, Label2)
17 end; (* WhileStatement *)
```

In line 5 a new label is defined and associated with the variable "*Label1*" and in line 6 the assembly directive "*DefAddr2*" is sent to the code file followed by the value of "*Label1*". This defines the label at the start of the loop.

In line 7 the symbol "while" is expected followed by symbols which can occur in expressions and statements, as well as the symbol "do" and the symbols with follow after a while statement.

In line 8 the procedure "*Expression*" is called. It recognizes and generates code for an expression. The follow set is similar to the previous instruction, with the symbols occurring in an expression removed from the follow set. The parameter "*ExprType*" returns the type of expression recognized.

In line 9 the procedure "*CheckTypes*" is used to check that the type of the expression, "*ExprType*", is the standard type boolean as the grammar rule specifies.

In line 10 the "do" symbol in the construct is expected followed by statement symbols and the follow set of the entire construct.

Line 11 defines a new label, "*Label2*", which is used in line 12 where the code instruction "*Do2*" is sent to the code file followed by the value of the label. This defines the jump for the case where the boolean expression evaluates to false.

In line 13 the procedure "*Pop*" is called with the value parameter 1, which signifies that one memory unit will be removed from the run time stack at this stage of the code execution.

In Line 14 the procedure "*Statement*" is called. "*Statement*" recognizes all statements and generates code for them.

Line 15 sends the pseudo code instruction "*Goto2*" to the intermediate code file, followed by the value of "*Label1*". This defines the repetition of the loop.

Line 16 sends the assembly directive "*DefAddr2*" to the intermediate code file, followed by the value of the label "*Label2*". This is used for defining the address to jump to if the boolean expression evaluates to false.

Line 17 is the end of the procedure "*WhileStatement*".

7.2.4 Exceptions.

The approach used to construct procedures for syntactic analysis and to add semantic analysis and code generation has been illustrated by the preceding examples. However, a number of exceptions should be discussed as they represent a different approach.

7.2.4.1. The When Clause.

In the draft international standard [ISO87] constraints are defined for constructs of Estelle. For the when clause the following constraint is given: (See Appendix A for the grammar rules for when clauses.)

> "The occurrence of an interaction-identifier in a when-clause shall constitute a defining-point of each interaction-argument-identifier in the interaction-argument- list associ-

ated with the interaction-identifier for the scope-region associated with the when-clause."

Unfortunately, together with the grammar rules and constraints defining transitions, the above constraint implies that, for a portion of a scope-region, interaction- argument-identifiers are visible. Because the scope is extended for only a portion of the scope-region, the method used in scope analysis can not be used. Instead of simply adding the interaction-argument-identifiers to the symbol table, it is necessary to remove the entries once the region associated with the when-clause is closed. To compound the agony, the syntax does not define an end to a when-clause, and this information is difficult to ascertain for a transition-group.

The problem was solved by first restricting a transition declaration so that only one transition block is permissible per transition declaration. In effect, all that the restriction does is to disallow the use of a short-hand notation for transitions. In addition, the single transition block's scope region is associated with the transition declaration, i.e. the transition block's scope region starts as soon as the transition declaration starts, and not when the transition block starts. Interaction-arguments- identifiers in a when-clause can then be uniquely defined for the scope region in which they are visible.

It is interesting to note the number of levels of derivation to obtain the interaction-argument-identifiers. From the interaction point obtained in the when-clause the channel and role are derived. The channel and role together are used to obtain the list of possible interactions, which are compared with the interaction-identifier in the when-clause to obtain an interaction pointer. The last-interaction- argument pointer, together with the length of the interaction arguments, is then used to redefine the interaction-arguments with the procedure "*Redefine*" which preserves the original information in symbol table entry, while making it visible at a higher level of the symbol table. The argument-identifier associated with the last- interaction-argument pointer is redefined and the length of the argument is subtracted from the total length of the arguments when the next-to-last interaction-argument's pointer is obtained from the current argument's "*Previous*" field. When the total length is zero after a subtraction, all the arguments have been redefined.

7.2.4.2. Clauses.

The when-clause was discussed in the previous section as an example where the scope scheme of the compiler had to be circumvented. In general the clauses of transition declaration together demonstrate an interesting use of the stop set in syntactic error recovery. The specification of Estelle defines the following for clauses of a transition declaration: (See Appendix A for the grammar rules. Note that, without the constraints, the grammar does not define what the specification claims to define for transition declarations.)

In a clause group only one clause of each type, i.e. provided-clause, from-clause, etc, may occur. In addition, in a clause group where a when-clause occurs, no delay- clause may occur, and vice versa. The clauses in a clause group may occur in any order. The restriction placed on the clause groups discussed in the previous section, namely that only one clause group is allowed per transition block means that some of the more complicated rules in the specification are ignored.

The problem to be solved is thus to allow a number of clauses in any order, but only one of each type. In the compiler the problem is solved by starting with all the possible symbols in the clauses, and then removing them from the stop set as soon as the relevant clause has occurred in the clause group. In this way syntactic errors can be generated when clauses are repeated, and the usual syntax error recovery and reported can be performed. It interesting to note that some of the conditions in the syntactic analysis change to accommodate this scheme. For example, a test
"**if** *Symbol* **in** *ClauseGroupSymbols* **then** ..." changes to
"**if** *Symbol* **in** (*Stop* $*$ *ClauseGroupSymbols*) **then** ...", i.e. not only must the symbol be a first symbol of a clause, but also in the stop set.

An additional problem caused by a combination of the arbitrary order of clauses and the extension of the scope region by the when clause is that the order of evaluation of the clauses is important. The following example from DIS 9074 [ISO87] illustrates the dependence problem:

1. **provided** $q > 1$;
   when $ip.m(q)$; and

2. **when** $ip.m(q)$;
   provided $q > 1$;

In the second case the provided-clause tests the parameter obtained in the when-clause. In the first case a variable which happens to have the same name as the parameter in the when-clause is tested. Because the when-clause opens the scope-region, it is important to distinguish between the two cases of evaluation order.

Fortunately, no ambiguity caused by the nomenclature arises, as the scheme used for the when-clauses allows a distinction between the two objects, even if the order of evaluation is changed. The parameter "*q*" and the variable "*q*" are at different levels of the symbol table and are therefore distinguished.

7.2.4.3. Modules.

The implementation of modules require some interesting deviations from the general scope scheme. For example, all the definitions in the module header must be included in the scope region of a module body associated with the module header. This inclusion is performed by redefinition similar to that used in when-clauses (See section 7.2.4.1.).

Unfortunately, the problem is compounded because interaction point definitions are defined without standard interaction point types. This means that between two records in the symbol table representing two interaction points there are records for defining the interaction point type. This problem also occurs for exported variable declarations. The records cannot be redefined by simply following the "*Previous*" field of the symbol table entry and decrementing the number of interaction points left to redefine, or in the case of the exported variables, subtracting the length of the variable from the total length. (See Figure 7.2.) The problem does not occur for the module parameter list as only named types are permissible for the parameter types.

The problem was solved by reorganizing the symbol table after the interaction points and exported variables have been defined in the module header definition. Redefinition can then occur without problems in the manner discussed in section 7.2.4.1. for when-clauses when a module body definition is compiled. The records used to define the type of the interaction point or exported variables are moved out of the direct link of records (See Figure 7.3.)

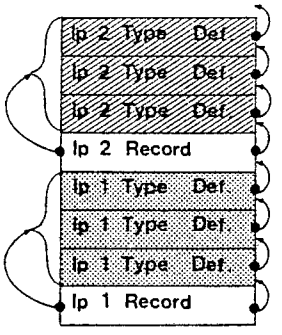

Figure 7.2: Symbol Table Fragment.

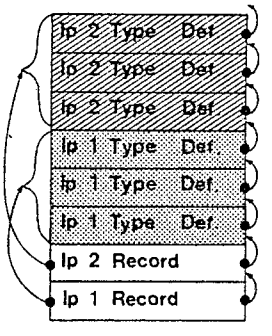

Figure 7.3: Symbol Table Fragment after Reorganization.

7.2.4.4. Procedures.

The use of semantic information in syntactic analysis has been discussed previously. (See sections 2.3.2. and 7.1.1.) An example is for procedure definitions where the identifier in the procedure definition can either be a previously defined procedure identifier or a new identifier for the procedure. In the latter case the procedure block can be given as well as the directives and a parameter list. In the former, only the procedure block can be given. Thus, two alternative syntactic constructs can be followed: either
< "procedure" *ProcedureIdentifier* ";" *ProcedureBlock* > or
< "procedure" *Identifier* ["(" *FormalParameterList* ")"] ";"
(*Directive* | *ProcedureBlock*) >.

By using the procedure "*FindDefinition*" the compiler ascertains whether a previous, incomplete definition was made of the procedure. In this case the first alternative is pursued. Note that the procedure "*Find*" can not be used, as the non-existence of a symbol table entry for the identifier is not an error per se. It is also impossible to first to declare the identifier and ascertain whether the entry is the entry defined, as error reporting of ambiguously defined identifiers is performed automatically.

In the next chapter Pass Three of the compiler, i.e. address resolution and code optimization is described.

Chapter 8

Pass Three

Pass Three is the final pass of the compiler. It has two functions, namely (1) to perform address resolution and (2) to perform some simple code optimization.

8.1 The Design of Pass Three.

Pass Three performs its two tasks in two passes of the intermediate code file generated by Pass Two. The first pass is used to collect references to addresses and set up a table mapping labels to true addresses. In this pass nothing is written to files. In the second pass true addresses are substituted for the labels, and certain sequences of instructions are replaced by instructions of an extended Estelle Machine.

Address resolution will be discussed next.

8.1.1 Address Resolution.

In Pass Two the directives "**DefAddr**" and "**DefArg**" were introduced. In Pass Three these directives are used in Pass Three's main task, namely address resolution. A table is kept of all the numerical values (i.e. labels) used to represent addresses, and when a "**DefArg**" is encountered, the table entry associated with the label is given a true

address. In the same way, when a "**DefAddr**" instruction is encountered, the current address is given as the true address associated with the label in the "**DefAddr**" directive.

Take for example the code produced for the while-statement "**while** B **do** S". Pass Two produces the following code:

DefAddr(17)
(* Evaluation code for B *)
if not B **then Jump**(18)
(* Code for S *)
Jump(17)
DefAddr(18)

Suppose the instructions of the while statement have the following addresses:

Address	**Code**
279	**DefAddr**(17)
279	(* Evaluation code for B *)
⋮	⋮
287	**if not** B **then Jump**(18)
289	(* Code for S *)
⋮	⋮
320	**Jump**(17)
322	**DefAddr**(18)

In the first pass of Pass Three, the two directives cause the following assignments:

$Table[17] := 279;$
$Table[18] := 322$

On the second pass the jump instructions, "**Do**(18)" and "**Goto**(17)", are replaced respectively by "**Do**(35)" and "**Goto**(−41)". The addresses 35 and −41 are really displacements from the current addresses. They are obtained as follows:

Address for the "**Do**" is $Table[18] - 287 = 322 - 287 = 35$
Address for the "**Goto**" is $Table[17] - 320 = 279 - 320 = -41$

The final code, which is sent to the code file is then:

(* Evaluation code for B *)
Do(35)
(* Code for S *)
Goto(−41)

This method is used for all the unresolved addresses in the intermediate code file. Note that the directives "**DefAddr**" and "**DefArg**" are not sent to the final code file, as they are not instructions of the Estelle Machine, but only directives for the Assembler (Pass Three).

8.1.2 Optimization.

Some peephole optimization is also done by Pass Three. The basis of the optimization is that some special cases of instructions occur frequently in the Estelle Code and can be replaced by instructions of an extended Estelle Machine. (The Estelle Machine is considered to be the machine with the instruction set which does have instructions for special cases of other instructions.) For example, the Estelle Code instruction "**Variable**(0, 13)" is an instruction often used to represent a local variable of a block. A new instruction, "**LocalVariable**", can be used which has one less argument.

At first eight optimization rules were defined, mapping special cases of instructions to instructions of the extended Estelle Machine:

1. $LocalVar(Displ) \equiv Variable(0, Displ)$
2. $LocalValue(Displ) \equiv Variable(0, Displ)\ Value(1)$
   or $LocalValue(Displ) \equiv LocalVar(Displ)\ Value(1)$
3. $GlobalVar(Displ) \equiv Variable(1, Displ)$
4. $GlobalValue(Displ) \equiv GlobalVar(Displ)\ Value(1)$
5. $SimpleValue \equiv Value(1)$
6. $SimpleAssign \equiv Assign(1)$
7. $GlobalCall(Displ) \equiv ProcCall(1, Displ)$
8. $Variable(Level, VarDispl + FieldDispl) \equiv$
   $Variable(Level, VarDispl)\ Field(FieldDispl)$

Optimization is performed by having procedures for each of the instructions that can be optimized. These procedures then generate

extended instructions and send them to the code file. Instructions, which are not in the set of instructions which have special cases which can be mapped to extended instructions, are just copied to the code file.

This method of optimization was used by Per Brinch Hansen in the Edison compiler [Bri82], where a reduction of in code size of 58% was achieved, and in the Pascal- compiler [Bri85], where a reduction of 32% was achieved. The reduction in the code size for Estelle is expected to be in the same order as the latter, as the reductions in code size reported for the former included the change to embedded line numbers, which is standard in the Pascal- code, as well as the Estelle code. Line numbers are included in procedure call instructions, etc., instead of being included with explicit line number updating instructions. The line numbers in question are used for run-time error-reporting.

The method used to define extended instructions for the Edison and Pascal- Machines was to examine the output of the compiler and find special cases of instructions which occurred frequently enough to warrant the loss of compilation speed. Presently the Estelle compiler performs the same optimizations as the Pascal- compiler, but should be extended, using the same methodology, when the Estelle is used extensively, allowing a clear picture to be formed of the code generated.

Other optimizations have been added without statisical analysis, namely the substitution of special forms of instructions such as "Multiply", "Divide" and "StringCompare".

8.1.3 Service Modules.

Pass Three uses services from three service modules, namely "*Administration*", "*Input*" and "*Output*". The modules, and the services they provide, are described in terms of the externally visible behaviour, and where of value, more information will be given.

8.1.3.1. "*Administration*".

"*Administration*" presents the services "*Emit*" and "*ReRun*". "*Emit*" is used to send final code to the code file. It uses a boolean, "*Emitting*", which inhibits output for the first pass. "*ReRun*" is used

to reopen the input file, (the intermediate code file generated by Pass Two), open the final code file, and by setting the boolean "*Emitting*", allows output to the final code file.

8.1.3.2. "*Input*".

"*Input*" provides the services "*NumberOfArguments*", "*ReadOp*", "*ReadArg*" and "*NextInstruction*". "*ReadOp*" and "*ReadArg*" are used to read the next instructions operation part, and an argument, respectively, from the intermediate code file. "*NumberOfArguments*" is used to find out how many arguments an instruction has. "*NextInstruction*" is used is read the next instruction by using "*ReadOp*", "*ReadArg*" and "*NumberOfArguments*".

8.1.3.3. "*Output*".

"*Output*" provides the services "*Emit1*", "*Emit2*", "*Emit3*" and "*Emit5*", which sends the ordinal value of an operator of the pseudo-code's representation to the final code file, via "*Emit*", along with 0, 1, 2 and 4 arguments respectively. These procedures also keep track of the address of the current instruction.

Note that, because "*Emit*" controls output, allowing output only on the assembler's second pass, the same procedures can be used for both passes. In the first pass any procedure calling "*Emit*", directly or indirectly, will have the output ignored. The same procedure, in the second pass, can send output to the code final using the same calls as the first pass.

8.1.4 "*Assemble and Optimize*".

The main module, "*Assemble and Optimize*", contains the functions "*Optimize*" and "*JumpDispl*" which indicate whether an instruction can be optimized, and what the relative displacement of a jump is, respectively. The procedures "*DefAddr*" and "*DefArg*" are used, respectively, to apply the directives of the same names. I.e. the arguments are entered into the address table to be used for address resolution.

While most of the procedures in the module will not be discussed as they are procedures used for optimization and are much the same, an example will be given. The procedure "*Field*" is used to remove the case where the argument of the "Field" instruction is zero. I.e. if the pseudo code/argument sequence "Field 0" occurs in the code, it is removed. In the case where the displacement is not zero the instruction/argument pair is just written to the code file.

The procedure "*CopyInstruction*" is used to copy an instruction and its arguments in the case where no optimization is performed for any instance of the instruction. "*Initialize*" is used to initialize the sets used to distinguish between instructions which can be optimized and those that are not. The procedure "*Assemble*" is used to pass through the code and perform the table entries when the directives are encountered, and to call the optimizing procedures.

The main program of Pass Three consists of a call to "*Administration*" which initializes the module "*Administration*", and prints an introductory message. "*Initialize*", which performs initialization as described, is then called, followed by the first pass through the code performed by "*Assemble*". "*ReRun*" is then called, followed by another pass performed by "*Assemble*". A final message is then printed.

The Estelle compiler environment includes the Estelle Machine, or Estelle Code interpreter. The machine will be described next, as well its implementation.

Chapter 9

Implementation of the Estelle Machine.

As was mentioned previously, the Estelle Machine is a hypothetical stack machine designed to provide the instructions needed to run Estelle Specifications. The Estelle Machine will be discussed, after which the implementation will be described.

9.1 The Design of the Estelle Machine.

For the purposes of this discussion the Estelle Machine will be divided into two components, namely the code interpreter and the execution manager. The code interpreter consists of the code necessary to interpret the E–code representation of a specification, while the execution manager consists of the code to control the execution of the interpreter, for example, the dynamic memory management, heap memory management and process management routines. The Estelle code interpreter will be discussed in detail, while the Estelle machine will be discussed to place the interpreter in context.

9.1.1 The Code Interpreter.

In essence the code interpreter consists of a number of registers and units to provide for instruction processing and data processing. Memory addressed by the interpreter consists of a fixed length area of code

and a stack area used to store variables. The stack area also contains temporary results during execution of instructions. The four registers accessed by the code interpreter are called "*ir*", "*br*", "*sr*" and "*sb*". "*ir*", the **instruction register**, contains the address of the current instruction; "*br*" the **base register**, is used to access variables. The **stack register** "*sr*" holds the address of the current top of the stack, while "*sb*" is used to store the address of the **bottom of the stack**. Note that the execution manager changes the current process executing by changing the registers of the code interpreter. The stack of interpreter is therefore not necessarily a fixed stack, but can actually be a number of separate stacks, one for each process executing. However, the stack will be discussed as if it were a single entity.

9.1.1.1. The Stack.

The stack of the Estelle Machine is initially empty. When specification execution begins, the Estelle Machine allocates space in the stack for variables defined in the specification block. These variables exist until specification execution ends. When the execution manager activates a process, the stack registers "*sr*" and "*sb*" are given values indicating the extent of the stack space which can be addressed. Thus, memory has been dynamically allocated to the process. The code interpreter now performs its own dynamic memory allocation within the stack space allocated by the execution manager.

When a procedure is activated the computer allocates space in the stack for the variables defined in the procedure block. These variables are removed when procedure execution ends. The variables of a single block are kept in a stack segment known as an "**activation record**" (AR). An AR consists of four parts: (The basic scheme underlying the method used has been discussed in section 3.4. See also Brinch Hansen [Bri85].)

(a) The parameter part.
(b) The context part.
(c) The variable part.
(d) The temporary part.

The parameter part contains storage for the formal parameters. The context part contains three addresses, (1) the **static link**, (2) the

dynamic link and (3) the **return address**. These addresses define the context in which the procedure was activated. The "*br*" register contains the address of the static link. This address is known as the **base address** of the AR. The variable part contains storage for the local variables. The temporary part contains operands and results during execution of statements. The parameter, context and variable part are of fixed length, while the temporary part is empty before, and after, procedure execution.

Within an AR, the parameters and local variables are placed in the order of definition. If a procedure is activated recursively, every activation creates another instance of the AR. It is therefore impossible to predict exact addresses of variables. Therefore the compiler uses the following method:

Every type in Estelle is a fixed type. Therefore every variable occupies a fixed number of words. This will be called the **length** of the variable. By combining the length of a variable with information concerning the order of the variables in the AR, it is possible to compute the relative address of every variable in the AR. These relative addresses are **displacements** relative to the base address of the AR.

On activation of a procedure, the interpreter creates an AR and makes the "*br*" register point to the base address of the record. Thus any variable in the AR can be accessed by adding its displacement to the value of the "*br*" register.

When a procedure terminates, the interpreter must remove the corresponding AR from the stack. To make this possible, the dynamic link of an AR contains the base address of the previous AR. When a procedure terminates, the dynamic link stored in the current AR is assigned to register "*br*". The chain of dynamic links that results is known as the **dynamic chain**.

The static links are used to define the set of variables that are accessible within the current block. This set of variables constitutes the context of the specification. As with dynamic links, a **static chain** can be defined. In general, every activation of a block may take place in a different context. Consequently, every AR is the start of a separate static chain. At any given moment, however, the current context is defined by a single static chain that starts at register "*br*".

9.1.1.2. Object Access.

To access an object in a context, the E–code must specify (1) the AR that contains the object, and (2) the displacement of the variable within the AR.

The compiler, during code generation, assigns a level number and a displacement to every variable. A level number is assigned to every block. The code that is generated, however, uses a relative level number. This is obtained by subtracting the level number of the variable from the level number of the current block. Access of a variable is performed by the code interpreter when it encounters an instruction "**Variable**(*Level*, *Displacement*)". This is done in five steps, say for the instruction "**Variable**(l, d)".

(a) The stack register "*sr*" is incremented by one to create a new temporary location on the top of the stack.

(b) The base address of the variable is found "l" levels down in the static chain.

(c) The absolute address of the variable is computed by adding the base address and the displacement "d".

(d) The absolute address is stored at the new temporary location.

(e) The instruction register "*ir*" is increment by 3 to make it point to the next instruction, 1 for the instruction's operation part, 1 for the level "l" and 1 for the displacement "d".

Variable instructions are used to access local variables and value parameters. Variable parameters are accessed by the instruction "**VarParam**(*Level*, *Displacement*)". This places the pointer at the absolute address calculated, onto the stack.

For an array variable "$A[j]$" the following instructions are executed by the computer: Where "LB_A" and "UB_A" define the lower and upper bounds of the array's index range, respectively, and "L_A" the length of a single element of the array. "l_A" and "l_j" define the levels of "A" and "j", and "d_A" and "d_j" their relative displacements.

1. **Variable**(l_A, d_A)
2. **Variable**(l_j, d_j)
3. **Value**(1)
4. **Index**(LB_A, UB_A, L_A, $LineNo$)

This process can be summarized as follows: (1) pushes the address of the array variable "A" onto the stack, (2) pushes the address of the index variable "j", (3) replaces the address of the index variable by its value, and (4) removes the index value from the stack, checks whether it is in the index range, and replaces the address of the array variable "A" by the address of the indexed variable "$A[j]$".

For field variables, variable access is performed as follows, say for "$x.g$",

1. **Variable**(l_x, d_x)
2. **Field**(d_g)

(1) places the address of "x" on the stack, while (2) add's "g" 's displacement to the address of "x" on the stack, to obtain the absolute address of "$x.g$".

Internal interaction points are accessed by the instructions

1. **Variable**(l_ip, d_ip)
2. **Ip**($LineNo$)

which place the address of the interaction point, which is found in the variable at the address "l_ip/d_ip", onto the stack and then verify that a valid interaction point reference is taking place.

A module variable, say "x", is treated in a similar way, with the instructions

1. **Variable**(l_x, d_x)
2. **ModuleVariable**($LineNo$).

While the compiler can verify that access to module variables is only performed by processes which may legally do so, the "**ModuleVariable**" instruction is necessary to make sure that the module variable has been initialized. The result of the instruction is to put the process identifier on the stack. The process identifier is, in fact, the base address of the process AR, "br".

Module parameters and exported variables are treated in the same way as value parameters for procedures if they are accessed from within the module. External interaction points are similarly treated as if they were value parameter interaction points. I.e. access is made to the address in the parameter list where the address of the interaction point

can be found as if it were a value parameter, and the "**ip**" instruction is used to verify the address on the stack. However, access to exported variables and external interaction points from parent processes requires more elaborate code. For example, the external interaction point "$B.b$" of a child process is accessed via the following instructions:

1. **Variable**(l_B, d_B)
2. **ModuleVariable**($LineNo$)
3. **Field**(d_b)
4. **Ip**($LineNo$)

While an external variable, say "x", of the same module will be accessed by the following instructions:

1. **Variable**(l_B, d_B)
2. **ModuleVariable**($LineNo$)
3. **Field**(d_x)

9.1.1.3. Expression Evaluation.

The code interpreter evaluates an expression in the temporary part of the current AR. The stack operands are accessed via the stack register "sr". A number of instructions are used for evaluation. Two instructions, namely "**Constant**($Value$)", and "**Value**($Length$)" (which has been mentioned before), are used to place values on the stack for evaluation. "**Constant**" places the value which is its argument on the stack. "**Value**" replaces the address on the stack by the value of length "**Length**".

Arithmetic and logical instructions that are available are "**Not**", "**Multiply**", "**Divide**", "**Modulo**", "**And**", "**Add**", "**Subtract**", "**Or**" and "**Minus**". Note that, in fact, a number of instructions occur which can be applied to real and integer arguments. However, differences are ignored for the purposes of the following summary. If "$St[sr]$" indicates the current stack top, then the execution of the instructions can be summarized as follows:

Not	$\equiv St[sr] := \textbf{not } St[sr]$
Multiply	$\equiv sr := sr - 1; St[sr] := St[sr] * St[sr+1]$
Divide	$\equiv sr := sr - 1; St[sr] := St[sr] \textbf{ div } St[sr+1]$
Modulo	$\equiv sr := sr - 1; St[sr] := St[sr] \textbf{ mod } St[sr+1]$
And	$\equiv sr := sr - 1; St[sr] := St[sr] \textbf{ and } St[sr+1]$
Add	$\equiv sr := sr - 1; St[sr] := St[sr] + St[sr+1]$
Or	$\equiv sr := sr - 1; St[sr] := St[sr] \textbf{ or } St[sr+1]$
Subtract	$\equiv sr := sr - 1; St[sr] := St[sr] - St[sr+1]$
Minus	$\equiv St[sr] := -St[sr]$

For the relational instructions, which are equally trivial, the following summarizes their execution:

Less	$\equiv sr := sr - 1; St[sr] := \textbf{ord}(St[sr] = St[sr+1])$
Equal	$\equiv sr := sr - 1; St[sr] := \textbf{ord}(St[sr] < St[sr+1])$
Greater	$\equiv sr := sr - 1; St[sr] := \textbf{ord}(St[sr] > St[sr+1])$
NotGreater	$\equiv sr := sr - 1; St[sr] := \textbf{ord}(St[sr] <= St[sr+1])$
NotEqual	$\equiv sr := sr - 1; St[sr] := \textbf{ord}(St[sr] <> St[sr+1])$
NotLess	$\equiv sr := sr - 1; St[sr] := \textbf{ord}(St[sr] >= St[sr+1])$

For example the code sequence for "$y * z$ **div** 5", would be executed by the Estelle Machine by executing the following instructions:

1. **Variable**(l_y, d_y)
2. **Value**(1)
3. **Variable**(l_z, d_z)
4. **Value**(1)
5. **Multiply**
6. **Constant**(5)
7. **Divide**

9.1.1.4. Statement Execution.

The assignment "$A := B$" is executed as follows:

1. **Variable**(l_A, d_A)
2. **Variable**(l_B, d_B)
3. **Value**(L_B)
4. **Assign**(L_A)

Where "L_A" and "L_B" are the lengths of the types of "A" and "B" respectively. (Note that due to semantic analysis "L_A" and "L_B" are

also equal.) The first 3 instructions have been discussed previously, and therefore not be discussed once more. "**Assign**" performs the assignment itself, placing the "L_B" values on top of the stack into the "L_A" words starting at the address in "$St[sr - L_B]$". If "A" and "B" were both integers, then the assignment is "$St[St[sr - 1]] := St[sr]$"

In a previous section the code for while– and if–statements was discussed. The instructions "**Goto**" and "**Do**" were, however, not discussed. They are control–flow instructions, respectively unconditional and conditional branches. The former, when executed, changes the instruction register "ir" to "$ir + Displ$", where "$Displ$" is a relative displacement from the "**Goto**($Displ$)" instruction. A "**Do**($Displ$)" instruction is similar, jumping to the displacement, or continuing to the next instruction, depending on the value on the top of the stack.

Compound statements, which are just a sequence of statement instructions, are trivial. Procedure statements are nontrivial, and need to be discussed in detail.

9.1.1.5. Procedure Activation.

An insight has already be gained into the use of AR's on the stack. The process involved in setting up the AR's has, however, not been discussed. This occurs during the execution of procedure call instructions and process creation instructions.

If an actual parameter exists for a procedure activation, instructions are executed to initialize these variables. For example, for a procedure activation "$P(x)$", the following instructions will be executed:

1. **Variable**(l_x, d_x)
2. **Value**(L_x)

These instructions create a new location in the temporary part of the stack, and assign the value of the previous instance of "x" to that location. (In the case of a function call, the instruction "**Push**(l)" will be executed, which places "l" words onto the stack to be used for the function's result.) Once these instructions have been executed, the computer executes a "**ProcCall**(l_P, d_P)" instruction. This first creates the AR and initializes the context there–of.

(a) The static link points to the AR of the surrounding block, and is found in the previous AR. (The context part there–of.) The "**ProcCall**" instruction defines the (relative) level in which the new static link is found.

(b) The dynamic link is the base address of the previous AR.

(c) The return address is the address of the first instruction after the "**ProcCall**" instruction. When the computer has executed the procedure, it uses this address to continue its execution.

After creating the context part of the AR, the computer switches execution to the procedure code. To make the code relocatable, the procedure code is defined by a displacement relative to the "**ProcCall**" instruction. The procedure is then executed when the computer finds the instruction "**Procedure**". The computer allocates space for the local variables of the procedure by incrementing the stack register "*sr*" by the variable length. It checks to see that there is sufficient space in the stack for temporaries and continues execution with the statement part of the procedure.

At the end of the procedure code, the computer executes the instruction "**EndProc**". It uses the dynamic link to remove the current AR. It then continues execution from the return address.

Note that standard procedures of Estelle are implemented as instructions of the Estelle Machine.

9.1.1.6. Estelle Primitives.

The Estelle primitives are also implemented as primitives of the Estelle Machine. E–code instructions are used to call the primitives with the correct parameters and to change the stack. The parameters are the arguments supplied with the E– code instruction associated with the primitive, together with values on the stack.

The init– and release–statements and the associated E–code instructions "**Init**" and "**Release**" concern process creation and the process hierarchy and will be discussed when processes are discussed.

For the connect–statement "**connect** *a* to *B.b*;" the following instructions will be executed:

1. **Variable**(l_a, d_a)
2. **Ip**(*LineNo*)
3. **Variable**(l_B, d_B)
4. **ModuleVariable**(*LineNo*)
5. **Field**(d_b)
6. **Ip**(*LineNo*)
7. **Connect**(0, 1, *LineNo*)

Instructions 1 to 6 place the addresses of the two interaction points onto the stack, while instruction 7 calls the Estelle Machine primitive "*connectx*($St[sr+1]$, $St[sr]$, 0, 1)". The parameters are thus the addresses of the two interaction points and a zero and a one indicating that the interaction points are internal and external respectively. For disconnect–statements the instruction "**Disconnect**(*LineNo*)" the primitive called is "*disconnectx*". The E–code instructions "**Connect**" and "**Disconnect**" decrement the stack register by two words after calling the primitive.

For attach– and detach–statements, instructions generated for the addresses of the interaction points will be followed by the instruction "**Attach**" or "**Detach**", respectively. On execution, these E–code instructions call the Estelle Machine primitives "*attachx*($St[sr+1]$, $St[sr]$)" and "*detachx*($St[sr+1]$, $St[sr]$)". "**Attach**" and "**Detach**" then decrement the stack register "*sr*" after the primitive has been called.

The code generated for an output–statement of the form
"**output** *ipx.ix*(*Interaction–argument–list*)"
is the following:

1. **Variable**(l_ipx, d_ipx)
2. **Ip**(*LineNo*)
3. **Constant**(*ix*)
4. (* Values of the interaction–arguments. *)
5. **Output**(*Length*, *LineNo*)

Where "*Length*" is the length, in words, of the interaction– argument–list. The first instruction will place the interaction point's address on the stack, the second will place the interaction identifier's internal representation onto the stack, and the instructions indicated by the comment will place the values of the actual parameters onto the

stack. "**Output**(*Length*, *LineNo*)" calls the Estelle Machine primitive "*outputx*(*St*[*sr* − *Length* − 1], *Length*)" which appends the interaction and interaction arguments occupying the top "*Length* + 1" words on the stack to the queue associated with the address in "*St*[*sr* − *Length* − 1]".

For transitions with when–clauses it is necessary to remove an interaction and its arguments from the queue once the transition has been fired. The instruction "**Input**(*Length*)" calls the Estelle Machine primitive "*inputx*(*St*[*sr*], *Length*)" which removes the interaction. Note that the address of the interaction point is placed on the stack with a simple "**Variable**" instruction, and "*inputx*" obtains the true interaction point itself. (It is unnecessary in this case to have an "**Ip**" instruction to access the interaction point, as we already know the interaction point must be valid, as we accessed it in the when–clause.)

9.1.1.7. Exist, All and Forone.

The domain–list form of the exist–expression and the all– and forone–statement will not be discussed, as the code consists of sequences of instructions described before.

The code generated for the example "**exist** *x*: *T* **suchthat** *B*" (where "*B*" is a boolean expression) was described in section 7.1.4.6. It is, however, necessary to examine the code in detail and it is therefore presented once more:

1. **Scan**(*ModuleType*, *LineNo*)
2. L_1: **Variable**(l_x, d_x)
3. **NextInstance**(L_2, *LineNo*)
4. (* Evaluation code for *B* *)
5. **Do**(L_1)
6. **Constant**(1)
7. L_2:

The first instruction, "**Scan**" is an Estelle Machine primitive which initializes the instance scan pointer [Dre87a] to the first module of "*ModuleType*". Subsequent calls to the "*nextinstancex*" primitive supplies the process identifier of the next instance of that module type. The E– code instruction "**NextInstance**" calls the primitive to obtain the next instance, and places the process identifier in the

address on top of the stack and removes the address from the stack. If "*nextinstancex*" returns a null process identifier, then "**NextInstance**" removes the address on top of the stack, places a zero on the stack, and jumps to the address indicated by the label L_2. (In that case, therefore the result of the exist–expression is false.) If the process identifier was not null, the code for the boolean expression is executed and the result of the boolean is tested by the next instruction which jumps to the address denoted by the label L_1, or proceeds to the next instruction, which places the value one on the stack, resulting in the exist– expression returning the value true.

All–statements are very similar. The code generated for the all–statement "**all** x: T **do** S" is as follows:

1. **Scan**(*ModuleType*, *LineNo*)
2. L_1: **Variable**(l_x, d_x)
3. **NextInstance**(L_2, *LineNo*)
4. (* Code for S *)
5. **Jump**(L_1)
6. L_2: **Pop**(1)

"**Scan**" and "**NextInstance**" are the same. The statement "**Pop**(1)" decrements the stack register "*sr*". The all– statement therefore behaves much like a for–statement, looping for all values of module instances.

The forone–statement is a combination of an if–statement and an exist–expression and will not be discussed further.

9.1.1.8. Transition Code.

The instruction which is generated for a transition, namely
"**Transition**(*From*, *To*, *Priority*, *Input*, L_1, L_2, L_3, *LineNo*)",
is used to set up a transition table in the Execution Manager. A transition will be initiated by the Execution Manager in the following manner: Naturally, the static information will be tested first, namely the from–set and the priority. The E–code instruction "**When**" is then called with argument L_1, (in the case of an input transition), followed by the E–code instruction "**Provided**" with parameter "L_2". (In the case of a spontaneous transition the E–code instruction "**Delay**" is called with parameter "L_1".) When one of these instructions is called, the return address is first placed on the stack, namely "$ir + 2$". Note

also that a dummy AR is also placed on the stack when transition evaluation is started. This makes provision for the transition block's variable's as well making provision for the way in which a when–clause extends the scope–region. In the case of when– claused transitions the instruction "**Input**(*Length*)" is executed, which removes the input interaction and its arguments from the queue.

The code produced for when–, delay– and provided–clauses was described in section 7.1.4.7. and will not be repeated, as the only E–code instruction not yet described is "**Return**". "**Return**(*Values*)" returns to the address which is at "$St[sr + Values]$". The "**When**", "**Delay**" and "**Provided**" E–code instructions start execution at the address given as a parameter and then evaluate the values on the stack, remove them and return to the return address now at the top the stack.

The transition for the initialization part of a module is very similar and will not be discussed further.

9.1.1.9. Module and Specification Code.

While the method in which AR's are created and removed within a process has been discussed, the creation and removal of processes themselves have not been discussed.

Because the Execution Manager supervises the process hierarchy, the code interpreter needs to know little about the processes themselves. Processes can therefore be represented as if they were procedures, i.e. with an AR. The creation of a process, which is performed by the Execution Manager, is initiated by the E–code instruction "**Init**".

"**Init**" is the E–code equivalent of the init–statement of Estelle. A typical sequence of E–code associated with the "**Init**" is the code generated for the init–statement
"**init** x **with** $y(1)$":
(Say "y" has one external interaction point.)

1. **Constant**(1)
2. **Variable**(l_x, d_x)
3. **Constant**(0)
4. **Init**(*ModNo*, *Vars*, *Tmp*, *InitL*, *TransL*, *Param*, *Exts*, *Exips*, *LineNo*)

Instruction 1. places the actual parameter of the module onto the stack, while 2. places the address of the module variable, which will be associated with the new process, onto the stack. 3. places a zero on the stack indicating that the queue discipline of the external interaction point is "common".

The "**Init**" instruction calls the Estelle Machine primitive "*initx*" with the same parameters. "*initx*" creates a new process control block, defines the queue discipline of the external interaction point, copies the parameters from the stack of the current process into the parameter space of the new process's stack, executes the code defining the internal interaction point's queue disciplines, initializes the queues associated with the process and performs the initialization code of the process. The process identification is then placed in the module variable defined by the address on the current stack, and the process is placed in the instance list of the module type "*ModNo*". The process identification is just the base register "*br*" of the new process. The code for the transition declarations is now executed to create a transition table. (Note that the arguments are removed from the current stack.)

The E–code instruction "**Release**", with the process identification on the stack, calls the primitive "*releasex*" with the process identification as parameter. "*releasex*" performs the release–statement for the process identification and removes the process identification from the stack.
"**Specification**(*Vars*, *Temps*, *InitL*, *TransL*, *LineNo*)"
is generated for specifications at the start of the code. The instruction is a simple form of "**Init**" and will not be discussed further.

Specification Execution.

After loading the specification, the code interpreter executes the first instruction of the specification. The first instruction of the specification is the instruction "**Specification**", which executes in much the same way as the "**Init**" instruction discussed above. The last instruction of a specification is an "**EndSpec**" instruction. This instruction is used to indicate to the process manager the the end of the specification process definition.

9.1.2 The Execution Manager.

As has been mentioned, the execution manager is the controlling unit of the Estelle Machine. Process management, which includes process scheduling, process memory allocation and inter–process communication, is the most important function performed, while heap memory management is also performed by the execution manager. While the execution manager will not be discussed, it is important to describe the mechanism used to communicate between the code interpreter and the execution manager.

The execution manager controls the instruction execution cycle by calling the code interpreter to interpret the next instruction. A process context switch is performed by changing the values of the interpreter's registers.

When an E–code instruction which concerns process existence, process communication, or heap memory allocation, is interpreted, procedures in the interface are called. The interpreter can therefore not change the process hierarchy, but must request these changes to be made. For example, via the primitives "*initx*", "*specificationx*" and "*releasex*".

9.2 The Implementation.

The Estelle Code Interpreter is the an implementation of the hypothetical Estelle Machine's code interpreter. A short description of the interpreter is all that is needed, as it closely resembles the hypothetical machine.

Each instruction defined for the Estelle Machine is implemented as a procedure. Note that on completion of an instruction, the associated procedure also updates the instruction register "*ir*" to point to the next instruction. It is important to note that the extended instructions of the Estelle Machine are also implemented as procedures of the interpreter.

The interpreter consists of a single module presenting services to the execution manager. The procedures "*StartMessage*" and "*EndMessage*" are used to print messages, while the procedure "*Error*" is used to print run time error messages. "*Initialize*" and

"*LoadSpecification*" are used to initialize all the variables of the interpreter, and read the specification into memory.

The core of the interpreter is the procedure "*NextInstruction*" which executes the current instruction pointed to by the instruction register "*ir*", and increments "*ir*". A boolean "*Running*" is set to true by the interpreter to allow "*RunSpecification*" to commence execution. When a run time error occurs, the procedure "*Error*" reports the error, and sets "*Running*" to false, thereby halting execution.

The rest of the interpreter consists of the procedures to implement the E–code instructions of the Estelle Machine. For example, the integer addition instruction "*Add*" is executed by the following procedure:

```
1.procedure Add;
2.begin
3.    sr := sr − 1;
4.    St[sr] := St[sr] + St[sr + 1];
5.    ir := ir + 1
6.end; (* Add *)
```

Line 3 decrements the stack by one word, (the size of an integer), while line 4 places the sum of the top most word (integer) and the word (integer) above, in the top most word of the stack. The decrementation of the stack register was done in line 3, so that the stack top contains the sum of the values. In line 5 the instruction register "*ir*" is incremented. Procedures for the other E–code instructions are similar.

In the next chapter of this document a number of concluding remarks are made concerning the compiler system development.

Chapter 10

Conclusion.

The value of the Estelle compiler project should be assessed on the basis of three questions, namely:

(a) How much was learnt about compilers?

(b) Can the Estelle compiler implemented be used?

(c) What was learnt concerning the FDT Estelle?

The first question is important as the project is the basis of a thesis for the fulfilment of the requirements for the degree of Master of Science in Computer Science. An evaluation of the Estelle compiler must therefore include comments on what was learnt by the implementor. The second question concerns the utility a tool designed and implemented, while the third is of interest as the Estelle compiler is one of the few Estelle language processors to be implemented to date.

The design and implementation of an Estelle compiler was a useful experience as the compilation of a non–trivial language was undertaken. The implementor was involved in all phases of compilation from lexical analysis to code generation, as well as the beginnings of a run time environment. Unfortunately, because of the not inconsiderable task of implementing a compiler for a large and complex language, the simplest solutions were used for every phase, and interesting methods could not be explored. The project was thus not a study into compilation methods, but rather the application of simple compilation methods to obtain a tool. The use of simplest solutions often caused problems. For example, the use of "on–the–fly" generation

of E–code caused problems with transitions, as code motion, associated with a more elaborate code generation scheme, was really needed to clear up the clumsy method used to evaluate when–, delay– and provided–clauses. One would also have liked to investigate aspects of compilation which were of interest, such as code generation and especially code optimization.

An interesting question is how one would approach the same project if one were to start again with present knowledge. The answer is undoubtly that the implementation of a well– defined run time environment should have been completed before work was started on the compiler. Unfortunately, the need for a processor for lexical, syntactic and semantic analysis of Estelle specification was the prime concern. The result of this ordering of tasks was that the implementor had, in fact, to design a run time environment "on–the–fly" during code generation. The design of a run time environment for a concurrent language such as Estelle can, in ordinary circumstances, by no means be considered an easy task.

The Estelle compiler system displays considerable utility and can be used for lexical, syntactic and semantic analysis for the entire Estelle. Code generation is performed for all but a few constructs of the language. Errors are detected, reported and recovered from in all stages of the compilation. Unfortunately, the implementation of a run time environment is needed to obtain a complete working system and to complete tests of the compiler system.

Further development needed on the compiler system includes the development of the run time environment, the addition of code generation for the remaining constructs of Estelle and the modification of the Estelle machine to provide for the meta–implementation. In addition, the Estelle machine, which is presently implemented in Pascal, should be re–implemented in Assembler as soon as the compiler system is fully operational. It is convenient at present, as it is easy to modify and expand, (although slower), in its Pascal form. Finally, the Estelle compiler should be extensively used, therefore making its development a useful exercise.

When one implements a language processor, one obtains a thorough knowledge of the language itself. Certain comments were made in section 2.3. and 2.4. concerning the advantages and disadvantages of features of Estelle. From an implementor's point of view the following

points were the most obvious comments about Estelle:

(a) Estelle is too big, i.e. it has unnecessary features. When a language is big, it implies that more code needs to be written, increasing the chance of errors. It is also frustrating to implement features which are unnecessary.

The inclusion of reals can be considered the most dubious decision. The application of reals in communication architecture software is, to say the least, not very common.

Another prime example of unnecessary features is the inclusion of the domain–list form of the exist–expression and all– and forone–statements which can not really be justified, as each can be implemented with a number of Pascal–like statements. However, one can understand that a compact notation is welcome for Estelle's role as a specification language.

(b) There are too many exceptions. A compiler implementor designs a certain phase in such a way that a general scheme is used for the greater part of the language, and exceptions are handled as special cases of the scheme. When a number of features are included in the language which do not adhere to the basic philosophy behind the language, the implementor's task is unenviable.

A prime example is the scope rules of Estelle. While the scope rules are block based, the when–clause adds interaction arguments to the scope which is the when–clause. Unfortunately, the when–clause does not occur between simple boundaries. For example, the transition declaration:

```
trans
provided B
   when ip.i(ia1, ia2, ..., ian)
      from S1
         to S3
1.          begin
               :
            end;

      from S2
         to S3
2.          begin
```

```
                ⋮
            end;

provided C
   from S1
      to S3
3.       begin
            ⋮
         end;
```

In transitions 1. and 2. the interaction arguments $ia_1 \ldots ia_n$ are visible in the transition block, but in 3. not. Because the provided–clause "**provided** B", which encloses the when– clause, is replaced by a new provided–clause, the interaction arguments are no longer visible. If a when– clause appears in, say, the clause part of transition 2., then the interaction arguments of that when–clause are visible in transition 2. This is all very messy. To compound the problem the any–clause can increase the amount of information which can be expressed in a number of transitions. While the problem can be solved, it should be remembered that if it is difficult to compile, it is also difficult to understand.

(c) Estelle is overly free.

The option ordering of clauses and short–hand notation in a transition declaration may make Estelle a better specification language, bringing it in line with the notational freedom of mathematics, but it certainly does little for understandibility. In addition, optional ordering of the declaration parts of modules adds complexity to the generation of code. This is also a form of exception, as the declaration parts of procedures and functions are defined to be in a certain order, simplifying matters. Estelle therefore allows an overly free style of specification which is not really of so much values as to justify the problems caused.

The absurdity of the free notational style which has been attempted is most clearly underlined by the fact that the clause groups of transitions are not adequately defined in the specification by grammar productions, but requires additional free text to resolve the ambiguities! While text is often used to note restrictions, there is a vast difference between the clause groups defined

by the grammar and the clauses groups defined when the text added.

In conclusion it can be said that fundamentally Estelle fails in that it is neither a specification language to rival more mathematical notations, or an implementation language to rival existing concurrent languages. However, it can play an important part in activities between specification and implementation, i.e. validation, testing and evaluation of communication protocols. The development of the Protocol Development Workbench is continuing with a scaled–down version of the Estelle compiler, as described, as the main component. Estelle features which were considered ill–advised, such as reals, the domain–list form of exist–expressions and forone– and all–statements, with–statements, optional ordering of declaration parts, etc, will be discarded to form a clean subset of Estelle.

Appendix E contains examples of output from the compiler with the original source text. The bibliography of this thesis, which follows, contains quoted references, as well as background literature.

Bibliography

[Aho72] Aho, A.V., and Ullman, J.D.,
The Theory of Parsing, Translation and Compiling – Volume 1: Parsing.,
Prentice–Hall, Englewood Cliffs, New Jersey,
ISBN 0–13–914556–7, 1972.

[Aho74] Aho, A.V., Johnson, S.C.,
LR Parsing.,
Computing Surveys,
Vol. 6, No. 2, pp. 99–124, 1974.

[Aho83] Aho, A.V., Hopcroft, J.E., Ullman, J.D.,
Data Structures and Algorithms.,
Addison–Wesley, Reading, Massachusetts,
ISBN 0–201–00023–7, 1983.

[Aho86] Aho, A.V., Sethi, R., Ullman, J.D.,
Compilers – Principles, Techniques, and Tools.,
Addison–Wesley, Reading, Massachusetts,
ISBN 0–201–10088–6, 1986.

[Al–86] Al–Hussaini, A.M.M., Stone, R.G.,
Yet Another Storage Technique for LR Parsing Tables.,
Software – Practice and Experience,
Vol. 16, No. 4, pp. 389–401, 1986.

[Amm77] Ammann, U.,
On Code Generation in a Pascal Compiler.,
Software – Practice and Experience,
Vol 7, pp. 391–423, 1977.

[And83] Andrews, G.R., Schneider, F.B.,
Concepts and Notations for Concurrent Program-

ming.,
Computing Surveys,
Vol. 15, No. 1, pp. 3–43, 1983.

[Ans87] Ansart, J.P., et al.,
Software Tools for Estelle.,
in Sarikaya and Bochmann [Sar87], pp. 55–61, 1988.

[Bau74] Bauer, F.L., Eickel, J., (Eds),
Lecture Notes in Computer Science Vol. 21: Compiler Construction – An Advanced Course.,
Springer–Verlag, New York,
ISBN 0–387–06958–5, 1974.

[Boc80] Bochmann, G.V.,
A General Transition Model for Protocols and Communication Services.,
IEEE Trans. on Communications,
Vol. Com–28, No. 4, pp. 643–650, 1980.

[Bri73] Brinch Hansen, P.,
Operating System Principles.,
Prentice–Hall, Englewood Cliffs, New Jersey,
ISBN 0–13–637843–9, 1973.

[Bri75] Brinch Hansen, P.,
The Programming Language Concurrent Pascal.,
IEEE Trans. on Software Engineering,
Vol. SE–1, No. 2, pp. 199–206, 1975.

[Bri78] Brinch Hansen, P.,
Distributed Processes: A Concurrent Programming Concept.,
Communications of the ACM,
Vol. 21, No. 11, pp. 934–941, 1978.

[Bri82] Brinch Hansen, P.,
Programming a Personal Computer.,
Prentice–Hall, Englewood Cliffs, New Jersey,
ISBN 0–13–730283–5, 1982.

[Bri85] Brinch Hansen, P.,
Brinch Hansen on Pascal Compilers.,
Prentice–Hall, Englewood Cliffs, New Jersey,
ISBN 0–13–730283–5, 1985.

[Bri87] Brinch Hansen, P.,
A Joyce Implementation.,
Software – Practice and Experience,
Vol. 17, No. 4, pp. 267–276, 1987.

[Bud88] Budkowski, S., Dembinski, P.,
An Introduction to Estelle: A Specification Language for Distributed Systems.
Computer Networks and ISDN Systems,
To be published, 1988.

[Cer88] Cerveira, A.G., (Ed),
Computer Communication Systems.,
Elsevier North–Holland, Amsterdam,
ISBN 0–444–70334–9, 1988.

[Cou87] Courtiat, J–P.,
Petri Net Based Semantics for Estelle.,
SEDOS Report SEDOS/109, 1987.

[Dan80] Danthine, A.A.S.,
Protocol Representation with Finite–State Models,
IEEE Trans. on Communications,
Vol. Com–27, No. 4, pp. 632–643, 1980.

[Day83] Day, J.D., Zimmerman, H.,
The OSI Reference Model.,
Proceedings of the IEEE,
Vol. 71, No. 12, pp. 1334–1340, 1983.

[Dei84] Deitel, H.M.,
An Introduction to Operating Systems.,
Addison–Wesley, Reading, Massachusetts,
ISBN 0–201–14502–2, 1984.

[Den84] Dencker, P., Durre, K., Heuft, J.,
Optimization of Parser Tables for Portable Compilers.,
ACM Trans. on Programming Languages and Systems,
Vol 6, No. 4, pp. 546–572, October 1984.

[DeR74] DeRemer, F.L.,
Lexical Analysis.
in Bauer and Eickel [Bau74], pp. 109–120, 1974.

[Dia86] Diaz, M., (Ed),
Protocol Specification, Testing and Verification V.,
Elsevier North–Holland, Amsterdam,
ISBN 0–444–87881–5, 1986.

[DoD80] United States Department of Defense,
Reference Manual for the Ada Programming Language.,
Proposed Standard Document, 1980.

[Dre87] Dreyer, J.J.,
Queues.,
Appendix to the Aide Memoire of the Eighth Meeting of the Data Network Architectures Laboratory,
Department of Computer Science,
University of Cape Town, 29 June 1987.

[Dre87a] Dreyer, J.J.,
Process Management.,
Preliminary notes,
Data Network Architectures Laboratory,
Department of Computer Science,
University of Cape Town, 1987.

[dSa87] de Saqui–Sanni, P., Courtiat, J–P.,
ESTIM: An Interpreter for the Simulation of Estelle Descriptions.,
SEDOS Report SEDOS/115, 1987.

[dSo88] de Souza, W.L., Ferneda, E.,
A Compiler for a Formal Description Technique.
in Cerveira [Cer88], pp. 275–283, 1988.

[Dij76] Dijkstra, E.W.,
A Discipline of Programming.,
Prentice–Hall, Englewood Cliffs, New Jersey,
ISBN 0–13–215871–X, 1976.

[Eng86] Engelbrecht, J.R., Kritzinger, P.S., Rudin, H.,
Predicting Protocol Performance from a Meta–Implementation.,
in Diaz [Dia86], pp. 349–362, 1986.

[Gil83] Gilbert, P.,
Software Design and Development.,

Science Research Associates, Chicago,
ISBN 0–574–21430–5, 1983.

[Gri71] Gries, D.,
Compiler Construction for Digital Computers.,
John Wiley, New York,
ISBN 0–471–32776–X, 1971.

[Gri81] Gries, D.,
The Science of Programming.,
Springer–Verlag, New York,
ISBN 0–387–90641–X, 1981.

[Gri74] Griffiths, M.,
LL(1) Grammars and Analysers.
in Bauer and Eickel [Bau74], pp. 57–84, 1974.

[Gri74a] Griffiths, M.,
Run–Time Storage Management.
in Bauer and Eickel [Bau74], pp. 195–221, 1974.

[Gro80] Grogono, P.,
Programming in Pascal.,
Addison–Wesley, Reading, Massachusetts,
ISBN 0–201–02775–5, 1980.

[Hoa85] Hoare, C.A.R.,
Communicating Sequential Processes.,
Prentice–Hall, Englewood Cliffs, New Jersey,
ISBN 0–13–153271–5, 1985.

[Hop79] Hopcroft, J.E., Ullman, J.D.,
Introduction to Automata Theory, Languages, and Computation.,
Addison–Wesley, Reading, Massachusetts,
ISBN 0–201–02988–X, 1979.

[Hor74] Horning, J.J.,
LR(1) Grammars and Analysers.
in Bauer and Eickel [Bau74], pp. 85–108, 1974.

[Hor74a] Horning, J.J.,
Structuring Compiler Development.
in Bauer and Eickel [Bau74], pp. 498–513, 1974.

[Hor74b] Horning, J.J.,
What the Compiler should tell the User.
in Bauer and Eickel [Bau74], pp. 525–543, 1974.

[Hor84] Horowitz, E.,
Fundamentals of Programming Languages. 2nd Ed,
Springer–Verlag, New York,
ISBN 3–540–12944–8, 1984.

[ISO85] International Organization for Standardization,
Information Processing Systems – Open Systems Interconnection – Estelle – A Formal Description Technique Based on an Extended State Transition Model,
Draft Proposal ISO/DP 9074, 1985.

[ISO86] International Organization for Standardization,
Information Processing Systems – Open Systems Interconnection – Estelle – A Formal Description Technique Based on an Extended State Transition Model,
2nd Draft Proposal ISO/DP 9074, 1986.

[ISO87] International Organization for Standardization,
Revised Text of ISO/DP 9074, Information Processing Systems – Open Systems Interconnection – Estelle – A Formal Description Technique Based on an Extended State Transition Model,
Draft International Standard DIS 9074, 1987.

[Jen78] Jensen, K., Wirth, N.
Pascal User Manual and Report. 2nd Ed,
Springer–Verlag, New York,
ISBN 0–387–90144–2, 1978.

[Knu73] Knuth, D.E.,
The Art of Computer Programming – Volume 3: Sorting and Searching.,
Addison–Wesley, Reading, Massachusetts,
ISBN 0–201–03803–X, 1973.

[Kri86] Kritzinger, P.S., van Dijk, J.,
Properties of Languages for the Implementation

of Communication Software Architectures.,
Technical Report CS-86-01-00,
Data Network Architectures Laboratory,
Department of Computer Science,
University of Cape Town, June 1986.

[Kri86a] Kritzinger, P.S.,
A Performance Model of the ISO Communication Architecture.,
IEEE Trans. on Communications,
Vol. Com-34, No. 6, pp. 554-563, 1986.

[Kri87] Kritzinger, P.S.,
Functional Description of the Protocol Development Workbench.,
Technical Report CS-87-02-00,
Data Network Architectures Laboratory,
Department of Computer Science,
University of Cape Town, January 1987.

[Lin86] Linn, R.J., et al.,
User Guide for the NBS Prototype Compiler for Estelle.,
Report No. ICST/APM 87-1,
U.S. National Bureau of Standards (ICST), November 1986.

[Lin86a] Linn, R.J. Jr,
A Tutorial on the Features and Facilities of Estelle.,
ISO/TC 97/SC 21/WG 1 Project 20, UMASS 2,
International Organization for Standardization, February 1986.

[Lin86b] Linn, R.J. Jr,
A Revised Draft Tutorial on the Features and Facilities of Estelle.,
ISO/TC 97/SC 21/WG 1 Project 20,
International Organization for Standardization, September 1986.

[McK74] McKeeman, W.M.,
Compiler Construction.
in Bauer and Eickel [Bau74], pp. 1-36, 1974.

[McK74a] McKeeman, W.M.,
Symbol Table Access.
in Bauer and Eickel [Bau74], pp. 253–301, 1974.

[Mil81] Miller, M.J., Linn, S.,
The Analysis of some Selective–repeat ARQ Schemes with Finite Receive Buffer.,
IEEE Trans. on Communications,
Vol. Com–29, No. 6, pp. 1307–1315, 1981.

[Par72] Parnas, D.L.,
On the Criteria to be used in Decomposing Systems into Modules.,
Communications of the ACM,
Vol 15, No. 12, pp. 1053–1058, 1972.

[Pem82] Pemberton, S., Daniels, M.C.,
Pascal Implementation: The P4 Compiler.,
Ellis Horwood, New York,
ISBN 0–85312–358–6, 1982.

[Pem82a] Pemberton, S., Daniels, M.C.,
Pascal Implementation: Compiler and Assembler/Interpreter.,
Ellis Horwood, New York,
ISBN 0–85312–437–X, 1982.

[Pet77] Peterson, J.L.,
Petri Nets,
Computing Surveys,
Vol. 9, No. 3, pp. 223–252, 1977.

[Poz82] Pozefsky, D.P., Smith, F.D.,
A Meta–Implementation for Systems Network Architecture.,
IEEE Trans. on Communications,
Vol. Com–30, No. 6, pp. 1348–1355, 1982.

[Pys80] Pyster, A.B.,
Compiler Design and Construction.,
Van Nostrand Reinhold, New York,
ISBN 0–442–24394–4, 1980.

[Ric85] Richter, A.,
Noncorrecting Syntax Error Recovery.,

ACM Trans. on Programming Languages and Systems,
Vol 7, No. 3, pp. 478–489, 1985.

[Roo86] Roos, J.,
The Protocol Specification Language Estelle.,
Quaestions Informatiae,
Vol 5, No. 1, pp. 51–62, 1986.

[Sar87] Sarikaya, B., Bochmann, G.V., (Eds),
Protocol Specification, Testing, and Verification VI.,
Elsevier North–Holland, Amsterdam,
ISBN 0–444–70293–8, 1987.

[Sch80] Schultz, G.D., et al.,
Executable Description and Validation of SNA.,
IEEE Trans. on Communications,
Vol. Com–28, No. 4, pp. 661–667, 1980.

[Sch82] Schwartz, R.L., Melliar–Smith, P.M.,
From State Machines to Temporal Logic: Specification Methods for Protocol Standards.,
IEEE Trans. on Communications,
Vol. Com–30, No. 12, pp. 2486–2496, 1982.

[Spe84] Spenke, M., et al.,
A Language Independent Error Recovery Method for LL(1) Parsers.,
Software – Practice and Experience,
Vol 14, No. 11, pp. 1095–1107, 1984.

[Sto82] Stotts Jr., P.D.,
A Comparative Study of Concurrent Programming Languages.,
ACM Sigplan Notices,
Vol. 17, No. 9, pp. 76–87, 1982.

[Ten81] Tennet, R.D.,
Principles of Programming Languages.,
Prentice–Hall, Englewood Cliffs, New Jersey,
ISBN 0–13–709873–1, 1981.

[Tre85] Tremblay, J–P., Sorenson, P.G.,
The Theory and Practice of Compiler Writing.,
McGraw–Hill, New York,
ISBN 0–07–065161–2, 1985.

[vDi87] van Dijk, J.,
Progress Report on Estelle Compiler System Development.,
Technical Report CS–87–01–00,
Data Network Architectures Laboratory,
Department of Computer Science,
University of Cape Town, January 1987.

[vDi87a] van Dijk, J.,
Second Progress Report on Estelle Compiler System Development.,
Technical Report CS–87–06–00,
Data Network Architectures Laboratory,
Department of Computer Science,
University of Cape Town, June 1987.

[Vis83] Vissers, C.A., Tenney, R.L., Bochmann, G.V.,
Formal Description Techniques.,
Proceedings of the IEEE,
Vol. 71, No. 12, pp. 1356–1364, 1983.

[Wai74] Waite, W.M.,
Semantic Analysis.
in Bauer and Eickel [Bau74], pp. 157–169, 1974.

[Wai74a] Waite, W.M.,
Code Generation.
in Bauer and Eickel [Bau74], pp. 302–332, 1974.

[Wai74b] Waite, W.M.,
Optimization.
in Bauer and Eickel [Bau74], pp. 549–602, 1974.

[Wai84] Waite, W.M., Goos, G.,
Compiler Construction.,
Springer–Verlag, New York,
ISBN 0–387–90821–8, 1984.

[Wai85] Waite, W.M., Carter, L.R.,
The Cost of a Generated Parser.,
Software – Practice and Experience,
Vol 15, No. 3, pp. 221–237, 1985.

[Weg83] Wegner, P., Smolka, S.A.,
Processes, Tasks, and Monitors: A Comparative

Study of Concurrent Programming Primitives.,
IEEE Trans. on Software Engineering,
Vol. SE–9, No. 4, pp. 446–462, 1983.

[Wes78] West, C.H.,
General Technique for Communication Protocol Validation.,
IBM J. Res. Develop.,
Vol. 22, No. 4, pp. 393–404, 1978.

[Wil84] Williamson, R., Horowitz, E.,
Concurrent Communication and Synchronization Mechanisms.,
Software – Practice and Experience,
Vol. 14, No. 2, pp. 135–151, 1984.

[Wir71] Wirth, N.,
The Design of a Pascal Compiler.,
Software – Practice and Experience,
Vol 1, pp. 309–333, 1971.

[Wir71a] Wirth, N.,
Program Development by Stepwise Refinement.,
Communications of the ACM,
Vol 14, No. 4, pp. 221–227, 1971.

[Wir73] Wirth, N.,
Systematic Programming – An Introduction.,
Prentice–Hall, Englewood Cliffs, New Jersey,
ISBN 0–13–880369–2, 1973.

[Wir76] Wirth, N.,
Algorithms + Data Structures = Programs.,
Prentice–Hall, Englewood Cliffs, New Jersey,
ISBN 0–13–022418–9, 1976.

[Wul81] Wulf, W.A., et al.,
Fundamental Structures of Computer Science.,
Addison–Wesley, Reading, Massachusetts,
ISBN 0–201–08824–X, 1981.

[Zim80] Zimmerman, H.,
OSI Reference Model – The ISO Model of Architectures for Open Systems Interconnection.,
IEEE Trans. on Communications,
Vol. Com–28, No. 4, pp. 425–432, 1980.

Appendix A

Specification of Estelle.

A.1 Specification of Lexical Analyzer for Estelle.

A.1.1 General.

The tokens used to construct Estelle programs can be classified into special-symbols, identifiers, directives, unsigned numbers, labels and charcter strings. The representation of any letter, upper-case or lower-case, differences of font, etc. occurring anywhere outside of a character string shall be insignificant in that occurrence to the meaning of the specification.

A.1.2 Basic Definitions.

Letters = "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|
"k"|"l"|"m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"|
"u"|"v"|"w"|"x"|"y"|"z"|"_".

Digits = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".

A.1.3 Special Symbols.

The special symbols are tokens having special meanings and shall be used to delimit the syntactic units of the language.

Special =	"+"	"−"	"*"	"/"	"="
	"<"	">"	"["	"]"	"."
	","	":"	";"	"^"	"("
	")"	"<>"	"<="	">="	":="
	".."	"..."	Word.		

Word =	"activity"	"all"	"and"
	"any"	"array"	"attach"
	"begin"	"body"	"by"
	"case"	"channel"	"common"
	"connect"	"const"	"default"
	"delay"	"detach"	"disconnect"
	"div"	"do"	"downto"
	"else"	"end"	"exist"
	"export"	"external"	"forone"
	"for"	"from"	"function"
	"goto"	"if"	"in"
	"individual"	"init"	"initialize"
	"ip"	"label"	"mod"
	"module"	"modvar"	"name"
	"not"	"of"	"or"
	"optional"	"otherwise"	"output"
	"packed"	"primitive"	"priority"
	"procedure"	"process"	"provided"
	"pure"	"queue"	"record"
	"release"	"repeat"	"same"
	"set"	"specification"	
	"state"	"stateset"	
	"suchthat"	"systemactivity"	"systemprocess"
	"then"	"timescale"	"to"
	"trans"	"type"	"until"
	"var"	"when"	"while"
	"with".		

A.1.4 Identifiers.

Identifiers may be any length. All characters of an identifier shall be significant in distinguishing between identifiers. No identifier shall have the same spelling as any word-symbol. Identifiers that are specified to be required shall have special significance, i.e. procedure, module, etc names.

Identifier = Letter { Letter | Digit }

Examples: x, *time*, *readinteger*, $tr_connect_eq$, $X1000$, X_1000, $\_x1000$, $InquireWorkStationTransformation$.

A.1.5 Directives.

The directives "forward", "external" and "primitive" shall be required directives. No directive shall have the same spelling as any word-symbol.

Directive = Letter { Letter | Digit }

Character-string	=	" ' " String-element { String-element }
String-element	=	Apostrophe-image | String-character
Apostrophe-image	=	" " "
String-character	=	Any character

Examples: 'A', ':', "", 'Estelle', '1111'.

Note: Conventionally, the apostrophe-image is regarded as a substitute for the apostrophe character, which cannot be a string character.

A.1.9 Token Separators.

The construct "{" any sequence of characters and separators of lines not containing the right brace "}" shall be a comment if the "}" does not occur within a character- string or within a comment. The substitution of a space for a comment shall not alter the meaning of the specification.

Comments, spaces (except in character-strings), and the separation of consecutive lines shall be considered to be token separators. Zero or more token separators may occur between any two consecutive tokens, or before the first token of a specification text. There shall be at least one separator between any pair of tokens made up of identifiers, word-symbols, labels or unsigned-numbers. No separators shall occur within tokens.

Examples: { This a comment }, { }.

A.1.10 Lexical Alternatives.

The representation for lexical tokens and separators given in the above shall constitute a reference representation for tokens and separators. The reference representation shall be used for specification interchange.

To facilitate the use of Estelle on processors that do not support the reference representation, the following alternatives have been defined.

All processors that have the required characters in their character set shall provide both the reference representations and the alternative representations; and the corresponding tokens or separators shall not be distinguished.

The alternative representations for the tokens shall be:

Reference token	Alternative token
↑	@
[	(.
]	.)

The Alternative forms of comment shall be all forms of a comment where one or both of the following substitutions are made:

Delimiting character	Alternative delimiting pairs of characters
{	(*
}	*)

A.2 Grammar of Estelle.

In some cases it is necessary to provide constraints to clarify the definition of certain constructs of the language. Informal semantics will, however, not be provided, as the constructs have been discussed in the preceeding document, notably in chapter 2.

The metalanguage used for the grammar are as follows:

=	:	Shall be defined to be.
|	:	Alternatively.
.	:	End of definition.
[x]	:	0 or 1 instance of x.
{ x }	:	0 or more instances of x.
[x]$^+$	:	1 or more instances of x.
(x | y)	:	Grouping: either x or y.
x $\uplus$ y	:	xy | yx.
$a_1 \uplus a_2 \uplus \ldots \uplus a_n$	:	All possible strings consisting of all the elements
	:	concatenated in an arbitrary order.
"xyz"	:	The terminal symbol xyz.

In addition, in the following section of the grammar, non-terminal symbols which are printed in bold type indicate that the non-terminal is defined in the Pascal subset of the language.

A.2.1 Estelle Grammar.

Specification = "specification" **Identifier** [SystemClass] ";"
[DefaultOptions]
[TimeOptions]
BodyDefinition
"end" ".".
SystemClass = "systemprocess" | "systemactivity".
DefaultOptions = "default" QueueDiscipline ";".
QueueDiscipline = "common" "queue" | "individual" "queue".
TimeOptions = "timescale" **Identifier** ";".

```
BodyDefinition = DeclarationPart
                 InitializationPart
                 TransitionDeclarationPart.
```

Note: Time units are hours, minutes, seconds, milliseconds and microseconds.

```
DeclarationPart = { Declarations }.
Declarations = ConstantDefinitionPart
             | TypeDefinitionPart
             | ChannelDefinition
             | ModuleHeaderDefinition
             | ModuleBodyDefinition
             | InteractionPointDeclarationPart
             | ModuleVariableDeclarationPart
             | VariableDeclarationPart
             | StateDefinitionPart
             | StateSetDefinitionPart
             | ProcedureAndFunctionDeclarationPart.
```

Note: The StateDefinitionPart may only occur once.

```
ChannelDefinition = ChannelHeading ChannelBlock.
ChannelHeading = "channel" Identifier "(" RoleList ")" ";".
ChannelIdentifier = Identifier.
RoleIdentifier = Identifier.
RoleList = Identifier "," Identifier.
ChannelBlock = { InteractionGroup }+.
InteractionGroup = "by" RoleIdentifier [ "," RoleIdentifier ] ":"
                   { InteractionDefinition }+.
InteractionDefinition = Identifier [ "(" ValueParameterSpecification
                        { ";" ValueParameterSpecification } ")" ] ";".
InteractionIdentifier = Identifier.

InteractionPointDeclarationPart = "ip" { InteractionPointDeclaration ";" }+.
InteractionPointDeclaration = IdentifierList ":" InteractionPointType.
                            | IdentifierList ":" "array" "[" IndexTypeList "]"
                                                 "of" InteractionPointType.
InteractionPointIdentifier = Identifier.
InteractionPointType = ChannelIdentifier "(" RoleIdentifier ")" [ QueueDiscipline ].
IndexTypeList = IndexType { "," IndexType }.

ModuleHeaderDefinition = "module" Identifier [Class] [ "(" ParameterList ")" ] ";"
                         [ "ip" { InteractionPointDeclaration ";" }+ ]
```

```
                        [ "export" { ExportedVariableDeclaration ";" }+ ]
                        "end" ";".
HeaderIdentifier = Identifier.
ParameterList = ValueParameterSpecification { ";" ValueParameterSpecification }.
ExportedVariableDeclaration = VariableDeclaration.
Class = "systemprocess" | "systemactivity" | "process" | "activity".

ModuleBodyDefinition = "body" Identifier "for" HeaderIdentifier ";"
                       ( BodyDefinition "end" ";" | "external" ";" ).
BodyIdentifier = Identifier.

ModuleVariableDeclarationPart = "modvar" { ModuleVariableDeclaration ";" }+.
ModuleVariableDeclaration = IdentifierList ":" HeaderIdentifier
                          | IdentifierList ":" "array" "[" IndexTypeList "]"
                                               "of" HeaderIdentifier.

StateDefinitionPart = "state" IdentifierList ";".
StateIdentifier = Identifier.

StateSetDefinitionPart = "stateset" { StateSetDefinition ";" }+.
StateSetDefinition = Identifier "=" StateSetConstant.
StateSetIdentifier = Identifier.
StateSetConstant = "[" StateIdentifier { "," StateIdentifier } "]".

ModuleVariable = ModuleVariableIdentifier
               | ModuleVariableIdentifier "[" IndexExpression
                                              { "," IndexExpression } "]".
ModuleVariableIdentifier = Identifier.

ChildExternalIp = ModuleVariable "." ExternalIp.
ConnectIp = ChildExternalIp | InternalIp.
ExternalIp = InteractionPointReference.
InternalIp = InteractionPointReference.
InteractionPointReference = InteractionPointIdentifier
                          [ "[" IndexExpression { "," IndexExpression } "]" ].

ExportedVariable = ModuleVariable "." ExportedVariableIdentifier.
ExportedVariableIdentifier = Identifier.

TransitionDeclarationPart = { TransitionDeclaration }.
TransitionDeclaration = "trans" TransitionGroup.
TransitionGroup = { ClauseGroup TransitionBlock ";" }+.
ClauseGroup = [ ProvidedClause ]
            ⊎ [ FromClause ]
```

⊎ [ToClause]
⊎ [AnyClause]
⊎ [DelayClause]
⊎ [WhenClause]
⊎ [PriorityClause].

TransitionBlock = **ConstantDefinitionPart**
TypeDefinitionPart
VariableDeclarationPart
ProcedureAndFunctionDeclarationPart
[TransitionName] **StatementPart.**

TransitionName = "name" **Identifier** ":".

Note: A transition block shall be well-formed, as defined below:

1. The category of a transition clause may be one of the following: provided, from, to, any, delay, when or priority.
2. If $C_1, \ldots, C_n$ $(n >= 1)$ are clauses of the same category C and $t_1, \ldots, t_n$ are well-formed transition-groups, then $C_1 t_1, \ldots, C_n t_n$ is a well-formed transition-group if and only if, the following three conditions are true:
   (a) (a) a clause of category C shall not appear in $t_1, \ldots, t_n$;
   (b) (b) If C is a when-clause category then a delay-clause does not appear in $t_1, \ldots, t_n$;
   (c) (c) If C is a delay-clause category then a when-clause does not appear in $t_1, \ldots, t_n$;

ToClause = "to" ToElement.
ToElement = "same" | StateIdentifier.

FromClause = "from" FromList.
FromList = FromElement { "," FromElement }.
FromElement = StateIdentifier | StateSetIdentifier.

ProvidedClause = "provided" (**BooleanExpression** | "otherwise").

WhenClause = "when" WhenIpReference "." InteractionIdentifier
[InteractionArgumentList].
WhenIpReference = InteractionPointIdentifier ["[" IpIndex { "," IpIndex } "]"].
IpIndex = **Constant** | **VariableIdentifier.**
InteractionArgumentIdentifier = **Identifier.**
InteractionArgumentList = "(" InteractionArgumentIdentifier
{ "," InteractionArgumentIdentifier }.

```
DelayClause = "delay" "(" ( Expression "," Expression
                          | Expression "," "*"
                          | Expression )
                    ")".

PriorityClause = "priority" PriorityConstant.
PriorityConstant = UnsignedInteger | ConstantIdentifier.

AnyClause = "any" DomainList "do".
DomainList = IdentifierList ":" OrdinalType
             { ";" IdentifierList ":" OrdinalType }.

InitializationPart = { "initialize" TransitionGroup }.

InitStatement = "init" ModuleVariable "with" BodyIdentifier
                [ "(" ActualModuleParameterList ")" ].
ActualModuleParameterList = ActualModuleParameter { "," ActualModuleParameter }.
ActualModuleParameter = Expression.

ReleaseStatement = "release" ModuleVariable.

ConnectStatement = "connect" ConnectIp "to" ConnectIp.

AttachStatement = "attach" ExternalIp "to" ChildExternalIp.

DisconnectStatement = "disconnect" ( ConnectIp | ModuleVariable ).

DetachStatement = "detach" ( ExternalIp | ChildExternalIp ).

OutputStatement = "output" InteractionReference [ ActualParameterList ].
InteractionReference = InteractionPointReference "." InteractionIdentifier.

AllStatement = "all" ( DomainList | ModuleDomain ) "do" Statement.
ModuleDomain = Identifier ":" HeaderIdentifier.

ForoneStatement = "forone" ( DomainList | ModuleDomain )
                  "suchthat" BooleanExpression
                  "do" Statement
                  [ "otherwise" Statement ].

ExistOne = "exist" ( DomainList | ModuleDomain ) "suchthat" BooleanExpression.
```

KeyWords	= "activity"	| "all"	| "any"
	| "attach"	| "body"	| "by"
	| "channel"	| "common"	| "connect"
	| "default"	| "delay"	| "detach"
	| "disconnect"	| "exist"	| "export"
	| "external"	| "forone"	| "from"
	| "individual"	| "init"	| "initialize"
	| "ip"	| "module"	| "modvar"
	| "name"	| "otherwise"	| "output"
	| "primitive"	| "priority"	| "process"
	| "provided"	| "pure"	| "queue"
	| "release"	| "same"	| "specification"
	| "state"	| "stateset"	| "suchthat"
	| "systemactivity"	| "systemprocess"	| "timescale"
	| "trans"	| "when".	

A.2.2 Additions to the Pascal Subset Grammar.

In this section "—" indicates the productions of the Pascal subset. Thus, "**Factor** = — | ExistOne." indicates that "ExistOne" should be added to the existing factors.

AssignmentStatement = — | ModuleVariable ":=" ModuleVariable.
ComponentVariable = — | ExportedVariable.
ConstantDefinition = — | **Identifier** "=" "any" **TypeIdentifier**.
Directive = **Letter** { **Letter** | **Digit** }.
Expression = — | ModuleVariable **RelationalOperator** ModuleVariable.
Factor = — | ExistOne.
FunctionHeading = — | "pure" "function" **Identifier**
[**FormalParameterList**] ":" **ResultType**.
FunctionIdentification =
— | "pure" "function" **FunctionIdentifier**.
Letter = — | "_".
ProcedureHeading = — | "pure" "procedure" **Identifier** [**FormalParameterList**].
ProcedureIdentification = — | "pure" "procedure" **ProcedureIdentifier**.
RepetitiveStatement = — | AllStatement.
ResultType = **TypeDenoter**.
SimpleStatement = — | AttachStatement
| ConnectStatement
| DetachStatement
| DisconnectStatement
| InitStatement
| OutputStatement
| ReleaseStatement.
StringCharacter = OneOfASetOfImplementationDefinedCharacters.

StructuredStatement = — | ForoneStatement.
TypeDefinition = — | **Identifier** "=" "...".
WordSymbol = — | KeyWords.

A.2.3 The Pascal Subset Grammar.

ActualParameter = Expression
| VariableAccess
| ProcedureIdentifier
| FunctionIdentifier.
ActualParameterList = "(" ActualParameter { "," ActualParameter } ")".
AddingOperator = "+" | "−" | "or".
ApostrapheImage = " " ".
ArrayType = "array" "[" **IndexType** { "," **IndexType** } "]" "of" ComponentType.
ArrayVariable = VariableAccess.
AssignmentStatement = (VariableAccess | FunctionIdentifier) ":=" Expression.

BaseType = OrdinalType.
Block = LabelDeclarationPart
ConstantDefinitionPart
TypeDefinitionPart
VariableDeclarationPart
ProcedureAndFunctionDeclarationPart
StatementPart.
BooleanExpression = Expression.

CaseConstant = Constant.
CaseConstantList = CaseConstant { "," CaseConstant }.
CaseIndex = Expression.
CaseListElement = CaseConstantList ":" Statement.
CaseStatement = "case" CaseIndex "of" CaseListElement
{ ";" CaseListElement } [";"] "end".
CharacterString = "'" StringElement { StringElement } "'".
ComponentType = TypeDenoter.
ComponentVariable = IndexedVariable | FieldDesignator.
CompoundStatement = "begin" StatementSequence "end".
ConditionalStatement = IfStatement | CaseStatement.
Constant = [Sign] (UnsignedNumber | ConstantIdentifier) | CharacterString.
ConstantDefinition = Identifier "=" Constant.
ConstantDefinitionPart = ["const" ConstantDefinition";" { ConstantDefinition ";" }].
ConstantIdentifier = Identifier.
ControlVariable = EntireVarible.
Digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
DigitSequence = Digit { Digit }.

```
DomainType = TypeIdentifier.

ElsePart = "else" Statement.
EmptyStatement = .
EntireVariable = VariableIdentifier.
EnumeratedType = "(" IdentifierList ")".
Expression = SimpleExpression [ RelationalOperator SimpleExpression ].

Factor = VariableAccess
       | UnsignedConstant
       | FunctionDesignator
       | SetConstructor
       | "(" Expression ")"
       | "not" Factor.
FieldDesignator = RecordVariable "." FieldSpecifier | FieldDesignatorIdentifier.
FieldDesignatorIdentifier = Identifier.
FieldIdentifier = Identifier.
FieldList = [ ( FixedPart [ ";" VariantPart ] | VariantPart ) [ ";" ] ].
FieldSpecifier = FieldIdentifier.
FinalValue = Expression.
FixedPart = RecordSection { ";" RecordSection }.
ForStatement = "for" ControlVariable ":=" InitialValue
               ( "to' | "downto" ) FinalValue "do" Statement.
FormalParameterList = "(" FormalParameterSection { ";" FormalParameterSection } ")".
FormalParameterSection = ValueParameterSpecification
                       | VariableParameterSpecification
                       | ProceduralParameterSpecification
                       | ValueParameterSpecification
                       | FunctionalParameterSpecification.
FractionalPart = DigitSequence.
FunctionBlock = Block.
FunctionDeclaration = FunctionHeading ";" Directive
                    | FunctionIdentification ";" FunctionBlock
                    | FunctionHeading ";" FunctionBlock.
FunctionDesignator = FunctionIdentifier [ ActualParameterList ].
FunctionHeading = FunctionIdentifier [ FormalParameterList ] ":" ResultType
FunctionIdentification = "function" FunctionIdentifier
FunctionIdentifier = Identifier.
FunctionParameterSpecification = FunctionHeading.

GotoStatement = "goto" Label.

IdentifiedVariable = PointerVariable "".
Identifier = Letter { Letter | Digit }.
```

```
IdentifierList = Identifier { ";" Identifier }.
IfStatement = "if" BooleanExpression "then" Statement [ ElsePart ].
IndexExpression = Expression.
IndexType = OrdinalType.
IndexedVariable = ArrayVariable "[" IndexExpression { ";" IndexExpression } "]".
InitialValue = Expression.

Label = DigitSequence.
LabelDeclarationPart = [ "label" Label ].
Letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" |
         "h" | "i" | "j" | "k" | "l" | "m" | "n" |
         "o" | "p" | "q" | "r" | "s" | "t" | "u" |
         "v" | "w" | "x" | "y" | "z".

MemberDesignator = Expression [ ".." Expression ].
MultiplyingOperator = "*" | "/" | "div" | "mod" | "and".

NewOrdinalType = EnumeratedType | SubrangeType.
NewPointerType = "" DomainType.
NewStructuredType = [ "packed" ] UnpackedStructuredType.
NewType = NewOrdinalType.
        | OrdinalStructuredType
        | NewPointerType.

OrdinalType = NewOrdinalType | OrdinalTypeIdentifier.
OrdinalTypeIdentifier = TypeIdentifier.

PointerType = NewPointerType | PointerTypeIdentifier.
PointerTypeIdentifier = TypeIdentifier.
PointerVariable = VariableAccess.
ProceduralParameterSpecification = ProcedureHeading.
ProcedureAndFunctionDeclarationPart = { ( ProcedureDeclaration
                                        | FunctionDeclaration ) ";" }.
ProcedureBlock = Block.
ProcedureDeclaration = ProcedureHeading ";" Directive
                     | ProcedureIdentification ";" ProcedureBlock
                     | ProcedureHeading ";" ProcedureBlock.
ProcedureHeading = "procedure" Identifier [ FormalParameterList ].
ProcedureIdentification = "procedure" ProcedureIdentifier.
ProcedureIdentifier = Identifier.
ProcedureStatement = ProcedureIdentifier [ ActualParameterList ].

RealTypeIdentifier = TypeIdentifier.
RecordSection = IdentifierList ":" TypeDenoter.
```

```
RecordType = "record" FieldList "end".
RecordVariable = VariableAccess.
RecordVariableList = RecordVariable { "," RecordVariable }.
RelationalOperator = "=" | "<>" | "<" | ">" | "<=" | ">=" | "in".
RepeatStatement = "repeat" StatementSequence "until" BooleanExpression.
RepetitiveStatement = RepeatStatement
                    | WhileStatement
                    | ForStatement.

ScaleFactor = SignedInteger.
SetConstructor = "[" [ MemberDesignator { "," MemberDesignator } ] "]".
SetType = "set" "of" BaseType.
Sign = "+" | "-".
SignedInteger = [ Sign ] UnsignedInteger.
SignedNumber = SignedInteger | SignedReal.
SignedReal = [ Sign ] UnsignedReal.
SimpleExpression = [ Sign ] Term { AddingOperator Term }.
SimpleStatement = EmptyStatement
                | AssignmentStatement
                | ProcedureStatement
                | GotoStatement.
SimpleType = OrdinalType | RealTypeIdentifier.
SimpleTypeIdentifier = TypeIdentifier.

    SpecialSymbol = "+"  | "-"  | "*"  | "/"  | "="
                  | "<"  | ">"  | "["  | "]"  | "."
                  | ","  | ":"  | ";"  | "^"  | "("
                  | ")"  | "<>" | "<=" | ">=" | ":="
                  | ".." | WordSymbol.

Statement = [ Label ":" ] ( SimpleStatement | StructuredStatement ).
StatementPart = CompoundStatement.
StatementSequence = Statement { ";" Statement }.
StringCharacter = OneOfASetOfImplementationDefinedCharacters.
StringElement = ApostropheImage | StringCharacter.
StructuredStatement = CompoundStatement
                    | ConditionalStatement
                    | RepetitiveStatement
                    | WithStatement.
StructuredType = NewStructuredType | StructuredTypeIdentifier.
StructuredTypeIdentifier = TypeIdentifier.
SubrangeType = Constant ".." Constant.

TagField = Identifier.
```

```
TagType = OrdinalTypeIdentifier.
Term = Factor { MultiplyingOperator Factor }.
TypeDefinition = Identifier "=" TypeDenoter.
TypeDefinitionPart = [ "type" TypeDefinition ";" { TypeDefinition ";" } ].
TypeDenoter = TypeIdentifier | NewType.
TypeIdentifier = Identifier.

UnpackedStructuredType = ArrayType | RecordType | SetType.
UnsignedConstant = UnsignedNumber
                 | CharacterString
                 | ConstantIdentifier
                 | "nil".
UnsignedInteger = DigitSequence.
UnsignedNumber = UnsignedInteger | UnsignedReal.
UnsignedReal = UnsignedInteger "." FractionalPart [ "e" ScaleFactor ]
             | UnsignedInteger "e" ScaleFactor.

ValueParameterSpecification = IdentifierList ":" TypeDenoter.
VariableAccess = EntireVariable | ComponentVariable | IdentifiedVariable.
VariableDeclaration = IdentifierList ":" TypeDenoter.
VariableDeclarationPart = [ "var" { VariableDeclaration ";" }+ ].
VariableIdentifier = Identifier.
VariableParameterSpecification = "var" IdentifierList ":" TypeIdentifier.
Variant = CaseConstantList ":" "(" FieldList ")".
VariantPart = "case" VariantSelector "of" Variant { ";" Variant }.
VariantSelector = [ TagField ":" ] TagType.

WhileStatement = "while" BooleanExpression "do" Statement.
WithStatement = "with" RecordVariableList "do" Statement.

WordSymbol = "and"    | "array"    | "begin"     | "case"
           | "const"  | "div"      | "do"        | "downto"
           | "else"   | "end"      | "for"       | "function"
           | "goto"   | "if"       | "in"        | "label"
           | "mod"    | "nil"      | "not"       | "of"
           | "or"     | "packed"   | "procedure" | "record"
           | "repeat" | "set"      | "then"      | "to"
           | "type"   | "until"    | "var"       | "while"
           | "with".
```

Appendix B

Standard Procedures and Functions.

This appendix contains the standard procedures and functions which are included in the Pascal subset of Estelle.

B.1 Required Standard Procedures.

B.1.1 Dynamic Allocation Procedures.

1. new(p)
   shall create a new variable that is totally undefined, shall create a new identifying-value of the pointer-type associated with p, that identifies the new variable, and shall attribute this identifying-value to the variable denoted by the variable-access p. The created variable shall possess the type that is the domain-type of the pointer-type possessed by p.

2. new($p, c_1, \ldots, c_n$)
   shall create a new variable that is totally undefined, shall create a new identifying-value of the pointer-type associated with p, that identifies the new variable, and shall attribute this identifying-value to the variable denoted by the variable-access p. The created variable shall possess the record-type that is the domain-type of the pointer-type possessed by p and shall have nested variants that correspond to the case-constants $c_1, \ldots, c_n$. The case- constants shall be listed in order of increasing nesting of the variant-parts. Any variant not specified

shall be at a deeper level of nesting than that specified by c_n. It shall be an error if a variant of a variant-part within the new variable becomes active and a different variant of the variant-part is one of the specified variants.

3. dispose(q)
   shall remove the identifying-value denoted by the expression q from the pointer-type of q. It shall be an error if the identifying-value had been created using the form **new**$(p, c_1, \ldots, c_n)$.

4. dispose($q, k_1, \ldots, k_m$)
   shall remove the identifying-value denoted by the expression q from the pointer-type of q. The case-constants $k_1, \ldots, k_m$ shall be listed in order of increasing nesting of the variant-parts. It shall be an error if the variable had been created using the form **new**$(p, c_1, \ldots, c_n)$ and m is not equal to n. It shall be an error if the variants in the variable identified by the pointer-value of q are different from those specified by the case-constants $k_1, \ldots, k_m$.

B.1.2 Transfer procedures.

Let a be a variable possessing a type that can be denoted by **array**$[s_1]$ **of** T, let z be a variable possessing a type that can be denoted by **packed array**$[s2]$ **of** T, and u and v be the smallest and largest values of the type s_2, then the statement "**pack**(a, i, z)" shall be equivalent to

```
begin
   k := i;
   for j := u to v do begin
      z[j] := a[k];
      if j <> v then k := succ(k)
   end
end
```

and the statement "**unpack**(z, a, i)" shall be equivalent to

```
begin
   k := i;
   for j := u to v do begin
      a[k] := z[j];
      if j <> v then k := succ(k)
```

```
    end
end
```

where j and k denote auxiliary variables that the specification does not otherwise contain. The type possessed by j shall be s_2, the type possessed by k shall be s_1, and i shall be an expression whose value shall be assignment-compatible with s_1.

B.2 Required Standard Functions.

B.2.1 Arithmetic functions.

For the following arithmetic functions, the expression x shall either of real-type or integer-type. For the functions **abs** and **sqr**, the result shall be the same as the type of the parameter, x. For the remaining arithmetic functions, the result shall always be of real-type.

Function	Result.
abs(x)	shall compute the absolute value of x.
sqr(x)	shall compute the square of x. It shall be an error if such a value does not exist.
sin(x)	shall compute the sine of x, where x is in radians.
cos(x)	shall compute the cosine of x, where x is in radians.
exp(x)	shall compute the value of the base of natural logarithms raised to the power x.
ln(x)	shall compute the natural logarithm of x, if x is greater than zero. It shall be an error if x is not greater than zero.
sqrt(x)	shall compute the non-negative square root of x, if x is not negative. It shall be an error if x is negative.
arctan(x)	shall compute the principle, in radians, of the arctangent of x.

B.2.2 Transfer Functions.

1. trunc(x).
   From the expression x that shall be of real-type, this function shall return a result of integer-type. The value of **trunc**(x) shall be such that if x is positive or zero then $0 <= x - \mathbf{trunc}(x) < 1$; otherwise

$-1 < x - \mathbf{trunc}(x) <= 0$. It shall be an error if such a value does not exist.

2. round(x).
From the expression x that shall be of real-type, this function shall return a result of integer-type. If x is positive or zero, $\mathbf{round}(x)$ shall be equivalent to $\mathbf{trunc}(x + 0.5)$, otherwise $\mathbf{round}(x)$ shall be equivalent to $\mathbf{trunc}(x - 0.5)$. It shall be an error if such a value does not exist.

B.2.3 Ordinal Functions.

1. ord(x).
from the expression x that shall be of an ordinal-type, this function shall return a result of integer-type that shall be the ordinal number of the value of the expression x.

2. chr(x).
from the expression x that shall be of integer-type, this function shall return a result of char-type that shall be the value whose ordinal number is equal to the value of the expression x if such a character value exists. It shall be an error if such a character value does not exist. For any value, ch, of char-type, it shall be true that:
$\mathbf{chr}(\mathbf{ord}(ch)) = ch$.

3. succ(x).
from the expression x that shall be of an ordinal-type, this function shall return a result that shall be of the same type as that of the expression. The function shall yield a value whose ordinal number is one greater than that of the expression x, if such a value exists. It shall be an error if such a value does not exist.

4. pred(x).
from the expression x that shall be of an ordinal-type, this function shall return a result that shall be of the same type as that of the expression. The function shall yield a value whose ordinal number is one less than that of the expression x, if such a value exists. It shall be an error if such a value does not exist.

B.2.4 Boolean Functions.

1. odd(x).
   From the expression x that shall be of of integer-type, this function shall be equivalent to $(\mathbf{abs}(x)\ \mathbf{mod}\ 2) = 1)$.

Appendix C

Error Messages.

C.1 Pass One Errors.

1. **Invalid Comment.**
   A comment is not closed properly.

2. **Invalid Integer.**
   An integer is too big. I.e. it is greater than 32767.

3. **Invalid Real.**
   A real is too big. I.e. the value is greater than 1.0e38, the exponent is greater than 38. Another error is a too big mantissa. Another error which causes an invalid real to be announced is a real which is not well-defined.

4. **Invalid String.**
   A string is not well-formed.

C.2 Pass Two Errors and Warnings.

The following are errors and warnings recognized by Pass Two of the Estelle compiler. Note that an error code provided by the compiler can be used to locate the exact instance of the error occurring. For example, if the error "124 Invalid Type" is reported, the code "124" can be used to find that the error condition was found in part one of five executable modules which make up Pass Two. (From the leading "1") The specific procedure call to the error reporting routine can then be found to be in the procedure "*Term*". A

number of type errors are found in the procedure, but "124" indicates that only the type "*boolean*" is acceptable for the multiplying operator "**and**". Another type error in the same procedure is "123", which indicates that the type "*integer*" is expected.

C.2.1 Pass Two Errors.

1. **Too Many Labels - Compilation Aborted.**
   When too many labels are generated for jumps, procedure labels, variable labels, etc, the compiler must abort as Pass Three's label tables are too small to accept so many labels. ("*MaxLabels*" is 1000.)

2. **Too Many Levels - Compilation Aborted.**
   The block table only makes provision for "*MaxLevel*" block records. I.e. there may only be "*MaxLevel*" levels of nesting in a specification. ("*MaxLevel*" is 10.)

3. **Ambiguous Identifier.**
   An identifier which has already be defined for that block is redefined.

4. **Types Incompatible in Assignment.**
   The expression on the right hand side of the assignment- statement can not be assigned to the variable on the left hand side.

5. **Invalid Non-local Control Variable.**
   A loop variable of a for-statement is not declared in the block closest containing the for-statement.

6. **Invalid Control Variable.**
   A loop variable of a for-statement is not a entire variable, i.e. the control variable is not indicated by a variable identifier.

7. **Incompatible Types.**
   Two types in an expression are not compatible.

8. **Invalid Exported Variable.**
   An invalid exported variable reference is attempted.

9. **Redefinition of Function's Parameters.**
   A function which was first defined with the "**forward**" directive is redefined with parameters.

10. **Invalid Assignment to Function Variable.**
    A function is assigned a value from outside the function's block.

11. **Too Many Case Indices.**
    A case-statement or variant record has more than "*MaxCaseIndices*" (500) case constants.

12. **Invalid Identifier Kind.**
    An identifier is of the wrong kind, e.g. it a type identifier, while a variable identifier was expected.

13. **Invalid Label Reference.**
    A goto-statement referenced a label which has not been defined.

14. **Invalid Assignment to Control Variable.**
    An assignment was made to the control variable of a for- statement.

15. **Missing Queue Discipline - No Default.**
    No queue discipline was specified, and no default value has been defined.

16. **Invalid Ordinal Type.**
    An ordinal type was expected.

17. **Invalid Pointer Type.**
    A pointer type was expected.

18. **Directive for Partial Definition.**
    A procedure or function which has already been defined with a directive is defined with a directive again.

19. **Redefinition of Procedure's Parameters.**
    A procedure which was first defined with the "**forward**" directive is redefined with parameters.

20. **Invalid Index Range.**
    A value outside the range of an index range has been found.

21. **Case Constant Repeated.**
    A case constant in a case-statement or variant record has been repeated.

22. **Role Repeated in Role List.**
    A role identifier has been repeated in a role list.

23. **Invalid Type for Selector.**
The selector for a case-statement or variant record is of the wrong type.

24. **Set Element Out of Range.**
An element of a set outside the range permissible for a set.

25. **Invalid Signed Constant.**
A constant which may not be signed has been signed.

26. **State Definition Repeated.**
More than one state definition part has been encountered in a module block.

27. **Invalid Syntax.**
The current input symbol does not conform to the grammar of the language.

28. **Missing System Class.**
A system class was needed for the module.

29. **Invalid Time Unit.**
A time unit other than the standard units has been used.

30. **Invalid Type.**
The type of an object is not of the type which was expected.

31. **Undefined Identifier.**
An identifier which is accessed in a block has not been defined.

32. **Label Reference Unresolved in Block.**
A label which was declared in a block and used in a goto-statement has not been resolved.

C.2.2 Pass Two Warnings.

1. **"any" Clause Not Implemented.**
The any-clause has not been implemented in this version of the compiler. (I.e. no code is generated.)

2. **"any Constant" Found - No Code Generated.**
The "any" form of constant definition was used. The specification is therefore not adequately specified.

3. **"..." Found - No Code Generated.**
   The "..." form of type definition was used. The specification is therefore not adequately specified.

4. **Field Designators Not Implemented.**
   As a result of with-statements not having been implemented, function designators were not implemented.

5. **"dispose" with Tags Not Implemented.**
   The standard procedure "dispose" was not implemented in the tag form. (The tagless form was implemented.)

6. **"external" Not Implemented.**
   This version of the compiler does not provide the directive "**external**" and therefore separate compilation.

7. **Functional Parameters Not Implemented.**
   Functional parameters for procedures and functions were not implemented for this version of the compiler.

8. **"new" with Tags Not Implemented.**
   The standard procedure "**new**" was not implemented in the tag form. (The tagless form was implemented.)

9. **"pack" Not Implemented.**
   The standard procedure "**pack**" was not implemented. A "**pack**" procedure has no effect.

10. **"packed" Not Implemented.**
    Packed types were not implemented.

11. **"primitive" Not Implemented.**
    This version of the compiler does not provide the directive "**primitive**".

12. **Procedural Parameters Not Implemented.**
    Procedural parameters for procedures and functions were not implemented for this version of the compiler.

13. **"unpack" Not Implemented.**
    The standard procedure "**unpack**" was not implemented. An "**unpack**" procedure has no effect.

14. **"with" Statement Not Implemented.**
    With-statements were not implemented.

C.3 Run Time Errors.

1. **No Statement for Case Expression.**
   The value obtain in the case expression doesn't correspond to any case constant.

2. **Set Range Error.**
   A value falls outside the range of a set.

3. **Range Error.**
   A value is not in a specified range.

4. **Stack Limit.**
   The stack has over- or under-flowed.

Appendix D

Restrictions.

This appendix contains a summary of restrictions placed on Estelle specifications by the Estelle compiler. A number of restrictions were placed on the language permissible:

D.1 Language Restrictions.

1. **Any-clauses are not implemented.**
   Any-clauses are not implemented. This partly due to the fact that their contribution is solely for notational purposes. In addition, due to the restrictions placed on clause groups in general, short-hand notations for clause groups are not possible.

2. **With-statements are not implemented.**
   As a with-statement is a mechanism which allows the programmer to explicitly perform code optimization, and not a feature which contributes readability to a specification, (on the contrary), their implementation is unnecessary. In addition, with-statements cause problems in code generation.

3. **Function designators are not implemented.**
   As a result of with-statements not having been implemented, function designators were not implemented.

4. **"new" and "dispose" were not implemented in the tag form.**
   As the tag form of "new" and "dispose" requires large amounts of supplementary code to be generated and more over- head during se-

mantic analysis, these forms were not implemented. (The tagless forms were implemented.)

5. **"external" and "primitive" were not implemented.**
   The directives "external" and "primitive" were not implemented as they are unnecessary for a first generation compiler.

6. **Functional and procedural parameters.**
   Functional and procedural parameters were not implemented as they add complexity. They were not necessary for this version of the compiler.

7. **"pack", "unpack" and "packed" are not supported.**
   The standard procedures "pack" and "unpacked" were not implemented. Once more there implementation would not seriously improve the value of the compiler. Similarly, the implementation of the "packed" directive for types was deemed unnecessary.

8. **Transitions.**
   A transition is restricted by allowing only one transition block transition declaration. In effect, all that the restriction does is to disallow the use of a short-hand notation for transitions. (This restriction is discussed in detail in section 7.2.4.1.) The short-hand notation is not really necessary, as the transitions can be explicitly enumerated. In addition, the rather dubious way in which the transition declarations had to be defined indicates that the simple notation should be used.

A number of restrictions are implementation specific: (They can be changed if the compiler is ported to another machine.)

D.2 Implementation Restrictions.

1. **Maximum size of an integer: 32767.**
   This is necessary as this the largest integer which can be represented in a single 16-bit word. The value of an integer can thus range from -32767 to 32767. One bit is reserved for the sign bit.

2. **Maximum size of a real: 1.0e38.**
   Once again the maximum size of reals are dictated by their representation in two consecutive 16-bit words. A real consists of a sign bit,

an exponent and the mantissa. The exponent can have a maximum value of 38.

3. **Maximum elements in a set: 64.**
   The number of elements is restricted by the number of 16-bit words used to represent sets. The elements can range in value from 0 to 63. The range is equivalent to the value range 0 to $16 * (SetSize - 1) - 1$, where "*SetSize*" is the constant 4. The elements of a set may only be in the range, and no provision is made for ranges such as 1 to 64. Sets were implemented because they were necessary for state sets. The common set concept was implemented as a by-product there-of.

4. **Maximum characters in a specification: 5000.**
   This restriction is placed by the size of the spelling table in Pass One. In fact it indicates that the total length of the characters in unique identifiers may not exceed 5000.

5. **Maximum length of an identifier: 80.**
   Although the specification of identifiers indicates that an identifier may be of arbitrary length, the restriction is placed as it is nonsensical for an identifier to longer than the length of a line. (A VDU allows the displaying of 80 characters in a line.)

6. **Maximum case constants: 500.**
   The restriction placed on the number of case indices prohibits the excessive generation of code for case- statements.

7. **Maximum levels of nesting: 10.**
   The maximum levels of nesting is restricted by the size of the block table, indicated by the constant "*MaxLevels*". The size of the block table is restricted to save memory.

8. **Maximum generated labels: 1000.**
   The number of labels generated during code generation is restricted by the size (MaxLabels) of the label tables in Pass Three.

Appendix E

Example Compilation.

In the various specification documents for Estelle the alternating bit protocol has been used to as an example of an Estelle specification. In this Appendix compilation is demonstrated for this traditional example: (Note that the "..." and "any constant" constructs used in the original example have been replaced with dummy types and valid constants.)

E.1 Alternating Bit Protocol Specification.

```
(* Alternating_Bit Example. From 2nd Dp 9074, September 1986, Version 0. *)
(*
   Specification: Alternating_Bit.
   Author       : ISO.
   Date         : 11/86.
   Version      : 0.
   Purpose      : This is a Specification of the Alternating Bit protocol
                  in the new Estelle (DP 9074, September 1986).
)

specification Alternating_Bit_Example systemprocess; timescale seconds;

(*
   This is the top level module body (specification)
   The specification has the attribute "systemprocess" and all its
   children (User, AB, Network) are processes. The time scale
   for delays is in seconds.
)
const
   Low = 0;                    (* Bounds of interaction point *)
```

```
   High = 1;                (* subscripts. *)
   Retran_Time = 2;         (* Retransmission time *)

type
   U_Data_type = record F1: integer end; (* User data. *)
   Seq_type = 0..1;                      (* Sequence number range. *)
   Id_type = (DATA, ACK);
   Cep_type = Low..High;
   Ndata_type =
      record
         Data: U_Data_type; (* User data. *)
         Id: Id_type;       (* Type of message. *)
         Conn: Cep_type;    (* Cep of sender. *)
         Seq: Seq_type      (* Sequence number. *)
      end;

(* Channel definitions for communication between processes *)

channel U_access_point(User, Provider);

   by User:
      SEND_request(Udata: U_Data_type);
      RECEIVE_request;

   by Provider:
      RECEIVE_response(Udata: U_Data_type);

channel N_access_point(User, Provider);

   by User:
      DATA_request(Ndata: Ndata_type);

   by Provider:
      DATA_response(Ndata: Ndata_type);

(* module header definitions *)

module User_type process
   (Conn_end_pt_id: Cep_type); (* Parameter list. *)
   ip                          (* interaction point list *)
   U: U_access_point(User) common queue;
end; (* of module header definition *)

(*
```

```
    The interaction point is named "U"; its channel_type
    is named "U_access_point" and the role of the module
    with respect to "U" is named "User". The queue of "U"
    is shared with other interaction points designated
    "common queue" within the module "User".
)

module Alternating_Bit_type process
    (Conn_end_pt_id: Cep_type); (* Parameter list. *)
    ip                          (* interaction point list *)
    U: U_access_point(Provider) common queue;
    N: N_access_point(User) individual queue;
end; (* of module header definition *)

(*
    The module has two interaction points named "U" and "N";
    the roles of the module are named:
    "Provider" with respect to "U", and
    "User" with respect to "N".
    Notice that there is an individual queue associated
    with the interaction point "N". If the queue would
    have been common (with the queue of "U") a
    SEND_request interaction point output by the user while
    the module is in the state Ack_Wait would lead to
    dead-lock (since the module would not be able to process
    a network DATA_response put in the same queue)
)

module Network_type process;
    ip                          (* interaction point list *)
    N: array[Cep_type] of
                                N_access_point(User) individual queue;
end; (* of module header definition *)

(*
    "N" is logically partioned into an array
    of identical interaction points; each may be
    identified by a subscript whose type is "Cep_type".
)

(* Body definitions for modules *)

body User_body for User_type; external;
```

```
body Network_body for Network_type; external;

(* The body for Alternating Bit is defined below: *)

body Alternating_Bit_body for Alternating_Bit_type;

type
   Msg_type =
      record
         MsgData: U_Data_type;
            MsgSeq: Seq_type
      end;
   Buffer_type = record F2: integer; M: Msg_type end;

var
   Send_buffer: Buffer_type;
   Recv_buffer: Buffer_type;
   Send_seq: Seq_type;
   Recv_seq: Seq_type;
   P: Msg_type;
   Q: Msg_type;
   B: Ndata_type;

state Ack_Wait, Estab; (* State definition part. *)

stateset (* Stateset definition part. *)
   Either = [Ack_Wait, Estab];

function Ack_ok(Nd: Ndata_type): boolean; primitive;

procedure Copy(var Todata: U_Data_type; Fromdata: U_Data_type);
(* Procedure provided by the implementer: Copy a user data variable *)
primitive;

procedure Empty(var Data: U_Data_type);
(* Procedure provided by the implementer: initialize a variable *)
(* holding user data to the value "no user data" *)
primitive;

procedure Format_data(Msg: Msg_type; var B: Ndata_type);
begin
   B.Id := DATA;
   B.Conn := Conn_end_pt_id;
   Copy(B.Data, Msg.MsgData);
```

```
    B.Seq := Msg.MsgSeq
end; (* Format_data *)

procedure Format_ack(Msg: Msg_type; var B: Ndata_type);
begin
    B.Id := ACK;
    B.Conn := Conn_end_pt_id;
    Empty(B.Data);
    B.Seq := Msg.MsgSeq
end; (* Format_ack *)

procedure Empty_buf(var Buf: Buffer_type); primitive;

procedure Store(var Buf: Buffer_type; Msg: Msg_type); primitive;

procedure Remove(var Buf: Buffer_type); primitive;

function Retrieve(Buf: Buffer_type): Msg_type; primitive;

function Buffer_empty(Buf: Buffer_type): boolean; primitive;

procedure Inc_send_seq;
begin Send_seq := (Send_seq + 1) mod 2 end;

procedure Inc_recv_seq;
begin Recv_seq := (Recv_seq + 1) mod 2 end;

initialize (* Initialization part of the Alternating_Bit process. *)

to Estab
    begin
        Send_seq := 0;
        Recv_seq := 0;
        Empty_buf(Send_buffer);
        Empty_buf(Recv_buffer);
    end;

trans
from Estab
    to Ack_Wait
        when U.SEND_request
            begin
                Copy(P.MsgData, Udata);
                P.MsgSeq := Send_seq;
```

```
            Store(Send_Buffer, P);
            Format_data(P, B);
            output N.DATA_request(B)
         end;

trans
from Either
   to same
      when U.RECEIVE_request
         provided not Buffer_empty(Recv_buffer)
            begin
               Q := Retrieve(Recv_buffer);
               output U.RECEIVE_response(Q.MsgData);
               Remove(Recv_buffer)
            end;

trans
from Ack_Wait
   to Ack_Wait
      delay(Retran_time)
         begin
            P := Retrieve(Send_buffer);
            Format_data(P, B);
            output N.DATA_request(B)
         end;

trans
from Ack_Wait
   to Estab
      when N.Data_response
         provided Ack_ok(Ndata)
            begin
               Remove(Send_Buffer);
               Inc_send_seq
            end;

trans
from Either
   to same
      when N.DATA_response
         provided Ndata.Id = DATA
            begin
               Copy(Q.MsgData, Ndata.Data);
               Q.MsgSeq := Ndata.Seq;
```

```
            Format_ack(Q, B);
            output N.Data_Request(B);
            if Ndata.Seq = Recv_seq then begin
               Store(Recv_buffer, Q);
               Inc_recv_seq
            end
         end;

end;

(* Module_variable_definition_part of the specification. *)

modvar
   User: array[Cep_type] of User_type;
   Alternating_Bit: array[Cep_type] of Alternating_Bit_type;
   Network: Network_type;

(* Initialization part of the specification. *)

initialize

begin
   init Network with Network_body;
   all Cep: Cep_type do begin
      (* Module initializations. *)
      init User[Cep] with User_body(Cep);
      init Alternating_Bit[Cep] with Alternating_Bit_body(Cep);
      (* Connection part. *)
      connect User[Cep].U to Alternating_Bit[Cep].U;
      connect Alternating_Bit[Cep].N to Network.N[Cep];
   end;
end;

(* End of specification. The specification has not transistion part. *)

end.
```

To envoke Pass One of the compiler the command "*ecpass1 a_b.est a_b.tok*" is given and the following appears on the screen:

```
Estelle Compiler Ver 3.0 Pass1: Scanner. J. van Dijk 1988
Pass1 completed
No errors detected
```

By making use of the utility "*tokens*", i.e. giving the command "*tokens* < *a_b.tok* > *t*", the contents of the file "*a_b.tok*" can be examined. (Note that, for this document, the file "*t*" was compacted by removing spaces and line-feeds. The original file was in the region of twenty pages long!)

E.2 Contents of Token File.

NewLine(1) *NewLine*(2) *NewLine*(3) *NewLine*(4) *NewLine*(5) *NewLine*(6)
NewLine(7) *NewLine*(8) *NewLine*(9) *NewLine*(10)
NewLine(11)
  Specification Identifier(25) *SystemProcess SemiColon*
  TimeScale Identifier(11) *SemiColon*
NewLine(12) *NewLine*(13) *NewLine*(14) *NewLine*(15) *NewLine*(16)
NewLine(17) *NewLine*(18) *NewLine*(19)
NewLine(20)
  Const
NewLine(21)
  Identifier(26) *Equal Integer*(0) *SemiColon*
NewLine(22)
  Identifier(27) *Equal Integer*(1) *SemiColon*
NewLine(23)
  Identifier(28) *Equal Integer*(2) *SemiColon*
NewLine(24)
NewLine(25)
  Type
NewLine(26)
  Identifier(29) *Equal Identifier*(26) *DoubleDot Identifier*(27)
  SemiColon
NewLine(27)
  Identifier(30) *Equal Record Identifier*(31) *Colon Identifier*(1)
  End SemiColon
NewLine(28)
  Identifier(32) *Equal Integer*(0) *DoubleDot Integer*(1) *SemiColon*
NewLine(29)
  Identifier(33) *Equal LeftParenthesis Identifier*(34) *Comma*
  Identifier(35) *RightParenthesis SemiColon*
NewLine(30)
  Identifier(36) *Equal*
NewLine(31)
  Record
NewLine(32)
  Identifier(37) *Colon Identifier*(33) *SemiColon*

NewLine(33)
Identifier(38) *Colon* *Identifier*(29) *SemiColon*
NewLine(34)
Identifier(34) *Colon* *Identifier*(30) *SemiColon*
NewLine(35)
Identifier(39) *Colon* *Identifier*(32)
NewLine(36)
End *SemiColon*
NewLine(37) *NewLine*(38) *NewLine*(39)
NewLine(40)
Channel *Identifier*(40) *LeftParenthesis* *Identifier*(41) *Comma*
Identifier(42) *RightParenthesis* *SemiColon*
NewLine(41)
NewLine(42)
By *Identifier*(41) *Colon*
NewLine(43)
Identifier(43) *LeftParenthesis* *Identifier*(44) *Colon* *Identifier*(30)
RightParenthesis *SemiColon*
NewLine(44)
Identifier(45) *SemiColon*
NewLine(45)
NewLine(46)
By *Identifier*(42) *Colon*
NewLine(47)
Identifier(46) *LeftParenthesis* *Identifier*(44) *Colon* *Identifier*(30)
RightParenthesis *SemiColon*
NewLine(48)
NewLine(49)
Channel *Identifier*(47) *LeftParenthesis* *Identifier*(41) *Comma*
Identifier(42) *RightParenthesis* *SemiColon*
NewLine(50)
NewLine(51)
By *Identifier*(41) *Colon*
NewLine(52)
Identifier(48) *LeftParenthesis* *Identifier*(49) *Colon* *Identifier*(36)
RightParenthesis *SemiColon*
NewLine(53)
NewLine(54)
By *Identifier*(42) *Colon*
NewLine(55)
Identifier(50) *LeftParenthesis* *Identifier*(49) *Colon* *Identifier*(36)
RightParenthesis *SemiColon*

NewLine(56) *NewLine*(57) *NewLine*(58)
NewLine(59)
Module Identifier(51) *Process*
NewLine(60)
LeftParenthesis Identifier(52) *Colon Identifier*(29) *RightParenthesis*
SemiColon
NewLine(61)
Ip
NewLine(62)
Identifier(53) *Colon Identifier*(40) *LeftParenthesis Identifier*(41)
RightParenthesis Common Queue SemiColon
NewLine(63)
End SemiColon
NewLine(64) *NewLine*(65) *NewLine*(66) *NewLine*(67) *NewLine*(68)
NewLine(69) *NewLine*(70) *NewLine*(71) *NewLine*(72)
NewLine(73)
Module Identifier(54) *Process*
NewLine(74)
LeftParenthesis Identifier(52) *Colon Identifier*(29) *RightParenthesis*
SemiColon
NewLine(75)
Ip
NewLine(76)
Identifier(53) *Colon Identifier*(40) *LeftParenthesis Identifier*(42)
RightParenthesis Common Queue SemiColon
NewLine(77)
Identifier(55) *Colon Identifier*(47) *LeftParenthesis Identifier*(41)
RightParenthesis Individual Queue SemiColon
NewLine(78)
End SemiColon
NewLine(79) *NewLine*(80) *NewLine*(81) *NewLine*(82) *NewLine*(83)
NewLine(84) *NewLine*(85) *NewLine*(86) *NewLine*(87) *NewLine*(88)
NewLine(89) *NewLine*(90) *NewLine*(91) *NewLine*(92) *NewLine*(93)
NewLine(94)
Module Identifier(56) *Process SemiColon*
NewLine(95)
Ip
NewLine(96)
Identifier(55) *Colon Array LeftBracket Identifier*(29) *RightBracket*
Of
NewLine(97)
Identifier(47) *LeftParenthesis Identifier*(41) *RightParenthesis*
Individual Queue SemiColon

NewLine(98)
End SemiColon
NewLine(99) *NewLine*(100) *NewLine*(101) *NewLine*(102) *NewLine*(103)
NewLine(104) *NewLine*(105) *NewLine*(106) *NewLine*(107)
NewLine(108)
Body Identifier(57) *For Identifier*(51) *SemiColon External*
SemiColon
NewLine(109)
NewLine(110)
Body Identifier(58) *For Identifier*(56) *SemiColon External*
SemiColon
NewLine(111) *NewLine*(112) *NewLine*(113)
NewLine(114)
Body Identifier(59) *For Identifier*(54) *SemiColon*
NewLine(115)
NewLine(116)
Type
NewLine(117)
Identifier(60) *Equal*
NewLine(118)
Record
NewLine(119)
Identifier(61) *Colon Identifier*(30) *SemiColon*
NewLine(120)
Identifier(62) *Colon Identifier*(32)
NewLine(121)
End SemiColon
NewLine(122)
Identifier(63) *Equal Record Identifier*(64) *Colon Identifier*(1)
SemiColon Identifier(65) *Colon Identifier*(60) *End SemiColon*
NewLine(123)
NewLine(124)
Var
NewLine(125)
Identifier(66) *Colon Identifier*(63) *SemiColon*
NewLine(126)
Identifier(67) *Colon Identifier*(63) *SemiColon*
NewLine(127)
Identifier(68) *Colon Identifier*(32) *SemiColon*
NewLine(128)
Identifier(69) *Colon Identifier*(32) *SemiColon*
NewLine(129)

Identifier(70) *Colon Identifier*(60) *SemiColon*
NewLine(130)
Identifier(71) *Colon Identifier*(60) *SemiColon*
NewLine(131)
Identifier(72) *Colon Identifier*(36) *SemiColon*
NewLine(132)
NewLine(133)
State Identifier(73) *Comma Identifier*(74) *SemiColon*
NewLine(134)
NewLine(135)
StateSet
NewLine(136)
Identifier(75) *Equal LeftBracket Identifier*(73) *Comma*
Identifier(74) *RightBracket SemiColon*
NewLine(137)
NewLine(138)
Function Identifier(76) *LeftParenthesis Identifier*(77) *Colon*
Identifier(36) *RightParenthesis Colon Identifier*(2) *SemiColon*
Primitive SemiColon
NewLine(139)
NewLine(140)
Procedure Identifier(78) *LeftParenthesis Var Identifier*(79) *Colon*
Identifier(30) *SemiColon Identifier*(80) *Colon Identifier*(30)
RightParenthesis SemiColon
NewLine(141)
NewLine(142)
Primitive SemiColon
NewLine(143)
NewLine(144)
Procedure Identifier(81) *LeftParenthesis Var Identifier*(34) *Colon*
Identifier(30) *RightParenthesis SemiColon*
NewLine(145) *NewLine*(146)
NewLine(147)
Primitive SemiColon
NewLine(148)
NewLine(149)
Procedure Identifier(82) *LeftParenthesis Identifier*(83) *Colon*
Identifier(60) *SemiColon Var Identifier*(72) *Colon Identifier*(36)
RightParenthesis SemiColon
NewLine(150)
Begin
NewLine(151)

Identifier(72) *Period Identifier*(37) *Becomes Identifier*(34) *SemiColon*
NewLine(152)
Identifier(72) *Period Identifier*(38) *Becomes Identifier*(52) *SemiColon*
NewLine(153)
Identifier(78) *LeftParenthesis Identifier*(72) *Period Identifier*(34)
Comma Identifier(83) *Period Identifier*(61) *RightParenthesis SemiColon*
NewLine(154)
Identifier(72) *Period Identifier*(39) *Becomes Identifier*(83) *Period*
Identifier(62)
NewLine(155)
End SemiColon
NewLine(156)
NewLine(157)
Procedure Identifier(84) *LeftParenthesis Identifier*(83) *Colon*
Identifier(60) *SemiColon Var Identifier*(72) *Colon Identifier*(36)
RightParenthesis SemiColon
NewLine(158)
Begin
NewLine(159)
Identifier(72) *Period Identifier*(37) *Becomes Identifier*(35) *SemiColon*
NewLine(160)
Identifier(72) *Period Identifier*(38) *Becomes Identifier*(52) *SemiColon*
NewLine(161)
Identifier(81) *LeftParenthesis Identifier*(72) *Period Identifier*(34)
RightParenthesis SemiColon
NewLine(162)
Identifier(72) *Period Identifier*(39) *Becomes Identifier*(83) *Period*
Identifier(62)
NewLine(163)
End SemiColon
NewLine(164)
NewLine(165)
Procedure Identifier(85) *LeftParenthesis Var Identifier*(86) *Colon*
Identifier(63) *RightParenthesis SemiColon Primitive SemiColon*
NewLine(166)
NewLine(167)
Procedure Identifier(87) *LeftParenthesis Var Identifier*(86) *Colon*
Identifier(63) *SemiColon Identifier*(83) *Colon Identifier*(60)
RightParenthesis SemiColon Primitive SemiColon
NewLine(168)

NewLine(169)
  Procedure Identifier(88) *LeftParenthesis Var Identifier*(86) *Colon*
  Identifier(63) *RightParenthesis SemiColon Primitive SemiColon*
NewLine(170)
NewLine(171)
  Function Identifier(89) *LeftParenthesis Identifier*(86) *Colon*
  Identifier(63) *RightParenthesis Colon Identifier*(60) *SemiColon*
  Primitive SemiColon
NewLine(172)
NewLine(173)
  Function Identifier(90) *LeftParenthesis Identifier*(86) *Colon*
  Identifier(63) *RightParenthesis Colon Identifier*(2) *SemiColon*
  Primitive SemiColon
NewLine(174)
NewLine(175)
  Procedure Identifier(91) *SemiColon*
NewLine(176)
  Begin Identifier(68) *Becomes LeftParenthesis Identifier*(68) *Plus*
  Integer(1) *RightParenthesis Mod Integer*(2) *End SemiColon*
NewLine(177)
NewLine(178)
  Procedure Identifier(92) *SemiColon*
NewLine(179)
  Begin Identifier(69) *Becomes LeftParenthesis Identifier*(69) *Plus*
  Integer(1) *RightParenthesis Mod Integer*(2) *End SemiColon*
NewLine(180)
NewLine(181)
  Initialize
NewLine(182)
NewLine(183)
  To Identifier(74)
NewLine(184)
  Begin
NewLine(185)
  Identifier(68) *Becomes Integer*(0) *SemiColon*
NewLine(186)
  Identifier(69) *Becomes Integer*(0) *SemiColon*
NewLine(187)
  Identifier(85) *LeftParenthesis Identifier*(66) *RightParenthesis SemiColon*
NewLine(188)
  Identifier(85) *LeftParenthesis Identifier*(67) *RightParenthesis SemiColon*

NewLine(189)
  End SemiColon
NewLine(190)
NewLine(191)
  Trans
NewLine(192)
  From Identifier(74)
NewLine(193)
  To Identifier(73)
NewLine(194)
  When Identifier(53) *Period Identifier*(43)
NewLine(195)
  Begin
NewLine(196)
  Identifier(78) *LeftParenthesis Identifier*(70) *Period Identifier*(61)
  Comma Identifier(44) *RightParenthesis SemiColon*
NewLine(197)
  Identifier(70) *Period Identifier*(62) *Becomes Identifier*(68) *SemiColon*
NewLine(198)
  Identifier(87) *LeftParenthesis Identifier*(66) *Comma Identifier*(70)
  RightParenthesis SemiColon
NewLine(199)
  Identifier(82) *LeftParenthesis Identifier*(70) *Comma Identifier*(72)
  RightParenthesis SemiColon
NewLine(200)
  Output Identifier(55) *Period Identifier*(48) *LeftParenthesis*
  Identifier(72) *RightParenthesis*
NewLine(201)
  End SemiColon
NewLine(202)
NewLine(203)
  Trans
NewLine(204)
  From Identifier(75)
NewLine(205)
  To Same
NewLine(206)
  When Identifier(53) *Period Identifier*(45)
NewLine(207)
  Provided Not Identifier(90) *LeftParenthesis Identifier*(67) *RightParenthesis*
NewLine(208)
  Begin

NewLine(209)
Identifier(71) *Becomes Identifier*(89) *LeftParenthesis Identifier*(67)
RightParenthesis SemiColon
NewLine(210)
Output Identifier(53) *Period Identifier*(46) *LeftParenthesis*
Identifier(71) *Period Identifier*(61) *RightParenthesis SemiColon*
NewLine(211)
Identifier(88) *LeftParenthesis Identifier*(67) *RightParenthesis*
NewLine(212)
End SemiColon
NewLine(213)
NewLine(214)
Trans
NewLine(215)
From Identifier(73)
NewLine(216)
To Identifier(73)
NewLine(217)
Delay LeftParenthesis Identifier(28) *RightParenthesis*
NewLine(218)
Begin
NewLine(219)
Identifier(70) *Becomes Identifier*(89) *LeftParenthesis Identifier*(66)
RightParenthesis SemiColon
NewLine(220)
Identifier(82) *LeftParenthesis Identifier*(70) *Comma Identifier*(72)
RightParenthesis SemiColon
NewLine(221)
Output Identifier(55) *Period Identifier*(48) *LeftParenthesis*
Identifier(72) *RightParenthesis*
NewLine(222)
End SemiColon
NewLine(223)
NewLine(224)
Trans
NewLine(225)
From Identifier(73)
NewLine(226)
To Identifier(74)
NewLine(227)
When Identifier(55) *Period Identifier*(50)
NewLine(228)

Provided Identifier(76) *LeftParenthesis Identifier*(49)
RightParenthesis
NewLine(229)
Begin
NewLine(230)
Identifier(88) *LeftParenthesis Identifier*(66) *RightParenthesis*
SemiColon
NewLine(231)
Identifier(91)
NewLine(232)
End SemiColon
NewLine(233)
NewLine(234)
Trans
NewLine(235)
From Identifier(75)
NewLine(236)
To Same
NewLine(237)
When Identifier(55) *Period Identifier*(50)
NewLine(238)
Provided Identifier(49) *Period Identifier*(37) *Equal Identifier*(34)
NewLine(239)
Begin
NewLine(240)
Identifier(78) *LeftParenthesis Identifier*(71) *Period Identifier*(61)
Comma Identifier(49) *Period Identifier*(34) *RightParenthesis*
SemiColon
NewLine(241)
Identifier(71) *Period Identifier*(62) *Becomes Identifier*(49) *Period*
Identifier(39) *SemiColon*
NewLine(242)
Identifier(84) *LeftParenthesis Identifier*(71) *Comma Identifier*(72)
RightParenthesis SemiColon
NewLine(243)
Output Identifier(55) *Period Identifier*(48) *LeftParenthesis*
Identifier(72) *RightParenthesis SemiColon*
NewLine(244)
If Identifier(49) *Period Identifier*(39) *Equal Identifier*(69) *Then*
Begin
NewLine(245)
Identifier(87) *LeftParenthesis Identifier*(67) *Comma Identifier*(71)
RightParenthesis SemiColon

NewLine(246)
  Identifier(92)
NewLine(247)
  End
NewLine(248)
  End SemiColon
NewLine(249) *NewLine*(250)
NewLine(251)
  End SemiColon
NewLine(252) *NewLine*(253) *NewLine*(254)
NewLine(255)
  ModVar
NewLine(256)
  Identifier(41) *Colon Array LeftBracket Identifier*(29) *RightBracket*
  Of Identifier(51) *SemiColon*
NewLine(257)
  Identifier(93) *Colon Array LeftBracket Identifier*(29) *RightBracket*
  Of Identifier(54) *SemiColon*
NewLine(258)
  Identifier(94) *Colon Identifier*(56) *SemiColon*
NewLine(259) *NewLine*(260) *NewLine*(261)
NewLine(262)
  Initialize
NewLine(263)
NewLine(264)
  Begin
NewLine(265)
  Init Identifier(94) *With Identifier*(58) *SemiColon*
NewLine(266)
  All Identifier(95) *Colon Identifier*(29) *Do Begin*
NewLine(267)
NewLine(268)
  Init Identifier(41) *LeftBracket Identifier*(95) *RightBracket With*
  Identifier(57) *LeftParenthesis Identifier*(95) *RightParenthesis SemiColon*
NewLine(269)
  Init Identifier(93) *LeftBracket Identifier*(95) *RightBracket With*
  Identifier(59) *LeftParenthesis Identifier*(95) *RightParenthesis SemiColon*
NewLine(270)
NewLine(271)
  Connect Identifier(41) *LeftBracket Identifier*(95) *RightBracket Period*
  Identifier(53) *To Identifier*(93) *LeftBracket Identifier*(95) *RightBracket*
  Period Identifier(53) *SemiColon*
NewLine(272)
  Connect Identifier(93) *LeftBracket Identifier*(95) *RightBracket Period*

Identifier(55) *To Identifier*(94) *Period Identifier*(55) *LeftBracket*
Identifier(95) *RightBracket SemiColon*
NewLine(273)
End SemiColon
NewLine(274)
End SemiColon
NewLine(275) *NewLine*(276) *NewLine*(277)
NewLine(278)
End Period
NewLine(279)
EndText

To envoke Pass Two of the compiler the command "*ecpass2 a_b.tok a_b.icf*" is given and the following appears on the screen:

```
Estelle Compiler Ver 3.0 Pass2: Parser. J. van Dijk 1988
434 Line 108 'External' Not Implemented
434 Line 110 'External' Not Implemented
300 Line 138 'Primitive' Not Implemented
300 Line 142 'Primitive' Not Implemented
300 Line 147 'Primitive' Not Implemented
300 Line 165 'Primitive' Not Implemented
300 Line 167 'Primitive' Not Implemented
300 Line 169 'Primitive' Not Implemented
300 Line 171 'Primitive' Not Implemented
300 Line 173 'Primitive' Not Implemented
Pass2 completed
```

By making use of the utility "pretty", i.e. giving the command "*pretty < a_b.icf > ic*", the contents of the file "*a_b.icf*" can be examined. (Note that the compiler does not indicate that no errors have occurred. This is a result of the warning messages generated. The specification is not fully defined as the directives "**external**" and "**primitive**" were used. However, for the purposes of this example the contents of the intermediate file is sufficient.

E.3 Intermediate Code.

```
Specification(1, 2, 3, 4, 20)
DefArg(5, 0)
DefArg(6, 0)
DefArg(7, 0)
DefArg(8, 0)
DefArg(9, 0)
DefArg(10, 0)
DefArg(11, 0)
DefArg(12, 0)
DefAddr(17)
Procedure(18, 19, 20, 138)
DefAddr(20)
DefArg(18, 0)
DefArg(19, 0)
EndProc(4)
DefAddr(21)
Procedure(22, 23, 24, 140)
DefAddr(24)
DefArg(22, 0)
DefArg(23, 0)
EndProc(2)
DefAddr(25)
Procedure(26, 27, 28, 144)
DefAddr(28)
DefArg(26, 0)
DefArg(27, 0)
EndProc(1)
DefAddr(29)
Procedure(30, 31, 32, 149)
DefAddr(32)
VarParam(0, -1)
Field(0)
Constant(0)
Assign(1)
VarParam(0, -1)
Field(1)
Variable(1, -3)
Value(1)
RangeAssign(0, 1, 1, 152)
VarParam(0, -1)
Field(2)
```

Variable(0, −3)
Field(0)
Value(1)
ProcCall(1, 21)
VarParam(0, −1)
Field(3)
Variable(0, −3)
Field(1)
Value(1)
RangeAssign(0, 1, 1, 155)
DefArg(30, 0)
DefArg(31, 5)
EndProc(3)
DefAddr(33)
Procedure(34, 35, 36, 157)
DefAddr(36)
VarParam(0, −1)
Field(0)
Constant(1)
Assign(1)
VarParam(0, −1)
Field(1)
Variable(1, −3)
Value(1)
RangeAssign(0, 1, 1, 160)
VarParam(0, −1)
Field(2)
ProcCall(1, 25)
VarParam(0, −1)
Field(3)
Variable(0, −3)
Field(1)
Value(1)
RangeAssign(0, 1, 1, 163)
DefArg(34, 0)
DefArg(35, 4)
EndProc(3)
DefAddr(37)
Procedure(38, 39, 40, 165)
DefAddr(40)
DefArg(38, 0)
DefArg(39, 0)

EndProc(1)
DefAddr(41)
Procedure(42, 43, 44, 167)
DefAddr(44)
DefArg(42, 0)
DefArg(43, 0)
EndProc(3)
DefAddr(45)
Procedure(46, 47, 48, 169)
DefAddr(48)
DefArg(46, 0)
DefArg(47, 0)
EndProc(1)
DefAddr(49)
Procedure(50, 51, 52, 171)
DefAddr(52)
DefArg(50, 0)
DefArg(51, 0)
EndProc(3)
DefAddr(53)
Procedure(54, 55, 56, 173)
DefAddr(56)
DefArg(54, 0)
DefArg(55, 0)
EndProc(3)
DefAddr(57)
Procedure(58, 59, 60, 175)
DefAddr(60)
Variable(1, 9)
Variable(1, 9)
Value(1)
Constant(1)
Add
Range(0, 1, 176)
Constant(2)
Modulo
Range(0, 1, 176)
RangeAssign(0, 1, 1, 176)
DefArg(58, 0)
DefArg(59, 3)
EndProc(0)
DefAddr(61)

Procedure(62, 63, 64, 178)
DefAddr(64)
Variable(1, 10)
Variable(1, 10)
Value(1)
Constant(1)
Add
Range(0, 1, 179)
Constant(2)
Modulo
Range(0, 1, 179)
RangeAssign(0, 1, 1, 179)
DefArg(62, 0)
DefArg(63, 3)
EndProc(0)
DefAddr(65)
Trans
Variable(1, 9)
Constant(0)
RangeAssign(0, 1, 1, 185)
Variable(1, 10)
Constant(0)
RangeAssign(0, 1, 1, 186)
Variable(1, 3)
ProcCall(1, 37)
Variable(1, 6)
ProcCall(1, 37)
EndTrans
DefAddr(15)
Initialize
Transition([], 1, 0, 0, 0, 0, 65, 184)
EndInitialize
DefAddr(66)
Variable(1, −2)
Ip(194)
Interaction(43, 194)
Return(1)
DefAddr(67)
Trans
Variable(1, 11)
Field(0)
Variable(1, −2)

Ip(196)
Field(1)
Value(1)
ProcCall(1, 21)
Variable(1, 11)
Field(1)
Variable(1, 9)
Value(1)
RangeAssign(0, 1, 1, 197)
Variable(1, 3)
Variable(1, 11)
Value(2)
ProcCall(1, 41)
Variable(1, 11)
Value(2)
Variable(1, 15)
ProcCall(1, 29)
Variable(1, −1)
Ip(200)
Constant(48)
Variable(1, 15)
Value(4)
Output(4, 201)
Variable(1, −2)
Input(1)
EndTrans
DefAddr(68)
Variable(1, −2)
Ip(206)
Interaction(45, 206)
Return(1)
DefAddr(69)
Push(1)
Variable(1, 6)
Value(3)
ProcCall(1, 53)
Not
Return(1)
DefAddr(70)
Trans
Variable(1, 13)
Push(2)

Variable(1, 6)
Value(3)
ProcCall(1, 49)
Assign(2)
Variable(1, −2)
Ip(210)
Constant(46)
Variable(1, 13)
Field(0)
Value(1)
Output(1, 210)
Variable(1, 6)
ProcCall(1, 45)
Variable(1, −2)
Input(0)
EndTrans
DefAddr(71)
Constant(2)
Copy
Return(2)
DefAddr(72)
Trans
Variable(1, 11)
Push(2)
Variable(1, 3)
Value(3)
ProcCall(1, 49)
Assign(2)
Variable(1, 11)
Value(2)
Variable(1, 15)
ProcCall(1, 29)
Variable(1, −1)
Ip(221)
Constant(48)
Variable(1, 15)
Value(4)
Output(4, 222)
EndTrans
DefAddr(73)
Variable(1, −1)
Ip(227)

Interaction(50, 227)
Return(1)
DefAddr(74)
Push(1)
Variable(1, −1)
Ip(227)
Field(1)
Value(4)
ProcCall(1, 17)
Return(1)
DefAddr(75)
Trans
Variable(1, 3)
ProcCall(1, 45)
ProcCall(1, 57)
Variable(1, −1)
Input(4)
EndTrans
DefAddr(76)
Variable(1, −1)
Ip(237)
Interaction(50, 237)
Return(1)
DefAddr(77)
Variable(1, −1)
Ip(238)
Field(1)
Field(0)
Value(1)
Constant(0)
Equal
Return(1)
DefAddr(78)
Trans
Variable(1, 13)
Field(0)
Variable(1, −1)
Ip(240)
Field(1)
Field(2)
Value(1)
ProcCall(1, 21)

Variable(1, 13)
Field(1)
Variable(1, −1)
Ip(241)
Field(1)
Field(3)
Value(1)
RangeAssign(0, 1, 1, 241)
Variable(1, 13)
Value(2)
Variable(1, 15)
ProcCall(1, 33)
Variable(1, −1)
Ip(243)
Constant(48)
Variable(1, 15)
Value(4)
Output(4, 243)
Variable(1, −1)
Ip(243)
Field(1)
Field(3)
Value(1)
Variable(1, 10)
Value(1)
Equal
Do(79)
Variable(1, 6)
Variable(1, 13)
Value(2)
ProcCall(1, 41)
ProcCall(1, 61)
DefAddr(79)
Variable(1, −1)
Input(4)
EndTrans
DefAddr(12)
Transition([1], 0, 0, 1, 66, 0, 67, 203)
Transition([0, 1], −1, 0, 1, 68, 69, 70, 214)
Transition([0], 0, 0, 0, 71, 0, 72, 224)
Transition([0], 1, 0, 1, 73, 74, 75, 234)
Transition([0, 1], −1, 0, 1, 76, 77, 78, 251)

EndTransition
DefArg(13, 16)
DefArg(14, 7)
DefAddr(80)
Trans
Variable(1, 6)
Constant(1)
Constant(1)
Init(3, 9, 10, 11, 12, 0, 0, 2, 267)
Variable(0, 3)
Constant(0)
Constant(1)
For(82, 1)
DefAddr(81)
Variable(0, 3)
Value(1)
Variable(1, 3)
Variable(0, 3)
Value(1)
Index(0, 1, 1, 268)
Constant(0)
Init(1, 5, 6, 7, 8, 1, 0, 1, 268)
Variable(0, 3)
Value(1)
Variable(1, 5)
Variable(0, 3)
Value(1)
Index(0, 1, 1, 269)
Constant(0)
Constant(1)
Init(2, 13, 14, 15, 16, 1, 0, 2, 269)
Variable(1, 3)
Variable(0, 3)
Value(1)
Index(0, 1, 1, 271)
ModuleVariable(271)
Field(−1)
Ip(271)
Variable(1, 4)
Variable(0, 3)
Value(1)
Index(0, 1, 1, 271)

ModuleVariable(271)
Field(−2)
Ip(271)
Connect(1, 1, 271)
Variable(1, 5)
Variable(0, 3)
Value(1)
Index(0, 1, 1, 272)
ModuleVariable(272)
Field(−1)
Ip(272)
Variable(1, 7)
ModuleVariable(272)
Field(−2)
Variable(0, 3)
Value(1)
Index(0, 1, 1, 272)
Ip(272)
Connect(1, 1, 272)
Next(81, 1)
DefAddr(82)
EndTrans
DefAddr(3)
Initialize
Transition([], −2, 0, 0, 0, 0, 80, 272)
EndInitialize
DefArg(4, 0)
DefArg(1, 6)
DefArg(2, 6)
EndSpec

To envoke Pass Three of the compiler the command "*ecpass3 a_b.icf a_b.ecf*" is given and the following appears on the screen:

Estelle Compiler Ver 3.0 Pass3: Assembler. J. van Dijk 1988
Pass3 completed

By making use of the utility "pretty", i.e. giving the command "*pretty < a_b.ecf > ec*", the contents of the file "*a_b.ecf*" can be examined. (Note that the compiler does not indicate that no errors have occurred. This is because no errors can occur at this stage of the compilation.

E.4 Final E-code.

```
0:   Specification(6, 6, 750, 0, 20)
6:   Procedure(0, 0, 5, 138)
11:  EndProc(4)
13:  Procedure(0, 0, 5, 140)
18:  EndProc(2)
20:  Procedure(0, 0, 5, 144)
25:  EndProc(1)
27:  Procedure(0, 5, 5, 149)
32:  VarParam(0, -1)
35:  Constant(0)
37:  SimpleAssign
38:  VarParam(0, -1)
41:  Field(1)
43:  GlobalVar(-3)
45:  SimpleValue
46:  RangeAssign(0, 1, 1, 152)
51:  VarParam(0, -1)
54:  Field(2)
56:  LocalValue(-3)
58:  GlobalCall(-45)
61:  VarParam(0, -1)
64:  Field(3)
66:  LocalValue(-2)
68:  RangeAssign(0, 1, 1, 155)
73:  EndProc(3)
75:  Procedure(0, 4, 5, 157)
80:  VarParam(0, -1)
83:  Constant(1)
85:  SimpleAssign
86:  VarParam(0, -1)
89:  Field(1)
91:  LocalValue(-3)
93:  RangeAssign(0, 1, 1, 160)
98:  VarParam(0, -1)
101: Field(2)
103: GlobalCall(-83)
105: VarParam(0, -1)
108: Field(3)
110: LocalValue(-2)
112: RangeAssign(0, 1, 1, 163)
117: EndProc(3)
```

119: **Procedure**(0, 0, 5, 165)
124: **EndProc**(1)
126: **Procedure**(0, 0, 5, 167)
131: **EndProc**(3)
133: **Procedure**(0, 0, 5, 169)
138: **EndProc**(1)
140: **Procedure**(0, 0, 5, 171)
145: **EndProc**(3)
147: **Procedure**(0, 0, 5, 173)
152: **EndProc**(3)
154: **Procedure**(0, 3, 5, 175)
159: **GlobalVar**(9)
161: **GlobalValue**(9)
163: **Constant**(1)
165: **Add**
166: **Range**(0, 1, 176)
170: **Constant**(2)
172: **Modulo**
173: **Range**(0, 1, 176)
177: **RangeAssign**(0, 1, 1, 176)
182: **EndProc**(0)
184: **Procedure**(0, 3, 5, 178)
189: **GlobalVar**(10)
191: **GlobalValue**(10)
193: **Constant**(1)
195: **Add**
196: **Range**(0, 1, 179)
200: **Constant**(2)
202: **Modulo**
203: **Range**(0, 1, 179)
207: **RangeAssign**(0, 1, 1, 179)
212: **EndProc**(0)
214: **GlobalVar**(9)
216: **Constant**(0)
218: **RangeAssign**(0, 1, 1, 185)
223: **GlobalVar**(10)
225: **Constant**(0)
227: **RangeAssign**(0, 1, 1, 186)
232: **GlobalVar**(3)
235: **GlobalCall**(−116)
237: **GlobalVar**(6)
239: **GlobalCall**(−120)

241: **EndTrans**
242: **Transition**([], 1, 0, 0, 0, 0, −28, 184)
254: **EndInitialize**
255: **GlobalVar**(−2)
257: **Ip**(194)
259: **Interaction**(43, 194)
262: **Return**(1)
264: **GlobalVar**(11)
266: **GlobalVar**(−2)
268: **Ip**(196)
270: **Field**(1)
272: **SimpleValue**
273: **GlobalCall**(−260)
275: **GlobalVar**(12)
279: **GlobalValue**(9)
281: **RangeAssign**(0, 1, 1, 197)
286: **GlobalVar**(3)
288: **GlobalVar**(11)
290: **Value**(2)
292: **GlobalCall**(−166)
294: **GlobalVar**(11)
296: **Value**(2)
298: **GlobalVar**(15)
300: **GlobalCall**(−273)
302: **GlobalVar**(-1)
304: **Ip**(200)
306: **Constant**(48)
308: **GlobalVar**(15)
310: **Value**(4)
312: **Output**(4, 201)
315: **GlobalVar**(−2)
317: **Input**(1)
319: **EndTrans**
320: **GlobalVar**(−2)
322: **Ip**(206)
324: **Interaction**(45, 206)
327: **Return**(1)
329: **Push**(1)
331: **GlobalVar**(6)
333: **Value**(3)
335: **GlobalCall**(−188)
337: **Not**

338: **Return**(1)
340: **GlobalVar**(13)
342: **Push**(2)
344: **GlobalVar**(6)
346: **Value**(3)
348: **GlobalCall**(−308)
350: **Assign**(2)
352: **GlobalVar**(−2)
354: **Ip**(210)
356: **Constant**(46)
358: **GlobalValue**(13)
360: **Output**(1, 210)
363: **GlobalVar**(6)
365: **GlobalCall**(−232)
367: **GlobalVar**(-2)
369: **Input**(0)
371: **EndTrans**
372: **Constant**(2)
374: **Copy**
375: **Return**(2)
377: **GlobalVar**(11)
379: **Push**(2)
381: **GlobalVar**(3)
383: **Value**(3)
385: **GlobalCall**(−245)
387: **Assign**(2)
389: **GlobalVar**(11)
391: **Value**(2)
393: **GlobalVar**(15)
395: **GlobalCall**(−368)
397: **GlobalVar**(-1)
399: **Ip**(221)
401: **Constant**(48)
403: **GlobalVar**(15)
405: **Value**(4)
407: **Output**(4, 222)
410: **EndTrans**
411: **GlobalVar**(−1)
413: **Ip**(227)
415: **Interaction**(50, 227)
418: **Return**(1)
420: **Push**(1)

422: GlobalVar(−1)
424: Ip(227)
426: Field(1)
428: Value(4)
430: GlobalCall(−424)
432: Return(1)
434: GlobalVar(3)
436: GlobalCall(−303)
438: GlobalCall(−284)
440: GlobalVar(−1)
442: Input(4)
444: EndTrans
445: GlobalVar(−1)
447: Ip(237)
449: Interaction(50, 237)
452: Return(1)
454: GlobalVar(−1)
456: Ip(238)
458: Field(1)
460: SimpleValue
461: Constant(0)
463: Equal
464: Return(1)
466: GlobalVar(13)
468: GlobalVar(−1)
470: Ip(240)
472: Field(1)
474: Field(2)
476: SimpleValue
477: GlobalCall(−464)
479: GlobalVar(13)
481: Field(1)
483: GlobalVar(−1)
485: Ip(241)
487: Field(1)
489: Field(3)
491: SimpleValue
492: RangeAssign(0, 1, 1, 241)
497: GlobalVar(13)
499: Value(2)
501: GlobalVar(15)
503: GlobalCall(−428)

505: GlobalVar(−1)
507: Ip(243)
509: Constant(48)
511: GlobalVar(15)
513: Value(4)
515: Output(4, 243)
518: GlobalVar(−1)
520: Ip(243)
522: Field(1)
524: Field(3)
526: SimpleValue
527: GlobalVar(10)
529: SimpleValue
530: Equal
531: Do(12)
533: GlobalVar(6)
535: GlobalVar(13)
537: Value(2)
539: GlobalCall(−413)
541: GlobalCall(−357)
543: GlobalVar(−1)
545: Input(4)
547: EndTrans
548: Transition([1], 0, 0, 1, −293, 0, −284, 203)
560: Transition([0, 1], −1, 0, 1, −240, −231, −220, 214)
572: Transition([0], 0, 0, 0, −200, 0, −195, 224)
584: Transition([0], 1, 0, 1, −173, −164, −150, 234)
596: Transition([0, 1], −1, 0, 1, −151, −145, −130, 251)
608: GlobalVar(6)
610: Constant(1)
612: Constant(1)
614: Init(3, 0, 0, 0, 0, 0, 0, 2, 267)
624: LocalVar(3)
626: Constant(0)
628: Constant(1)
630: For(119, 1)
633: LocalValue(3)
635: GlobalVar(3)
637: LocalValue(3)
639: Index(0, 1, 1, 268)
644: Constant(0)
646: Init(1, 0, 0, 0, 0, 1, 0, 1, 268)

656: **LocalValue**(3)
658: **GlobalVar**(5)
660: **LocalValue**(3)
662: **Index**(0, 1, 1, 269)
664: **Constant**(0)
666: **Constant**(1)
668: **Init**(2, 16, 7, −426, −120, 1, 0, 2, 269)
678: **GlobalVar**(3)
680: **LocalValue**(3)
682: **Index**(0, 1, 1, 271)
687: **ModuleVariable**(271)
689: **Field**(−1)
691: **Ip**(271)
693: **GlobalVar**(4)
695: **LocalValue**(3)
697: **Index**(0, 1, 1, 271)
702: **ModuleVariable**(271)
704: **Field**(−2)
706: **Ip**(271)
708: **Connect**(1, 1, 271)
712: **GlobalVar**(5)
714: **LocalValue**(3)
716: **Index**(0, 1, 1, 272)
721: **ModuleVariable**(272)
723: **Field**(−1)
725: **Ip**(272)
727: **GlobalVar**(7)
729: **ModuleVariable**(272)
731: **Field**(−2)
733: **LocalValue**(3)
735: **Index**(0, 1, 1, 272)
740: **Ip**(272)
742: **Connect**(1, 1, 272)
746: **Next**(−113, 1)
749: **EndTrans**
750: **Transition**([], −2, 0, 0, 0, 0, −142, 272)
762: **EndInitialize**
763: **EndSpec**

Appendix F

The Debugger.

The debugger used in testing the Estelle machine, and which will be used as the basis of a simulation system, is associated with a Command Line Interpreter (CLI). The grammar of the command language will be given, as well as the function of commands specified by the command language.

F.1 Debugger CLI Grammar.

```
Register = "br" | "ir" | "sb" | "sr".
Factor = IntegerConstant
       | RealConstant
       | BooleanConstant
       | CharacterConstant
       | "(" Expr ")"
       | Register
       | "st" "[" Expr "]"
       | ( "bool" | "boolean" ) "(" Expr ")"
       | ( "character" | "char" ) "(" Expr ")"
       | ( "int" | "integer" "(" Expr ")"
       | "real" "(" Expr ")".
MultiplyingOperator = "*" | "/" | "div".
Term = Factor [ MultiplyingOperator Factor ].
AddingOperator = "+" | "-".
Expr = Term [ AddingOperator Term ]*.
RegCommand = ( "register" | "reg" ) Register [ ":" Expr ].
StackCommand = "stack" [ Expr [ ( ":" | "," ) Expr ] ].
StepCommand = "step" [ [ Expr "," ] Expr ].
PrintBoolean = ( "bool" | "boolean" ) Expr "," Expr.
PrintChar = ( "char" | "character" ) Expr "," Expr.
```

```
PrintInteger = ( "int" | "integer" ) Expr "," Expr.
PrintReal = "real" Expr "," Expr.
Command = PrintBoolean [";"]
        | PrintChar [";"]
        | PrintInteger [";"]
        | PrintReal [";"]
        | RegCommand [";"]
        | StackCommand [";"]
        | StepCommand [";"]
        | "quit" [";"].
```

F.2 Command Summary.

While the above specifies the language recognized by the CLI, a brief discussion is necessary before the functionality of commands is discussed.

The primary purpose of the debugger is to make it possible to observe the behaviour of an executing specification. To this end commands are provided which allow the printing of the stack, registers and a trace of executing E-code instructions. To make the debugger more useful parameters have been added to allow the modification of values that have been examined, as well as allowing values in any part of the stack to be examined or modified, as well as any portion of the code to be traced. The parameters can take the form of simple expressions utilizing the adding operations "+" and "−" and the multiplication operations "*", "div" and "/". Expressions can contain registers i.e. the values of the registers "br", "ir", "sb" and "sr", stack values (indicated by "st" "[" Expr "]"), integer-, real-, boolean- and character constants, as well as stack values interpreted as integer-, real-, boolean- and character constants, for example, "bool(sr)". Stack elements and a range of stack elements can also be printed interpreted as one of the standard types. The commands follow in detail:

F.2.1 PrintBoolean.

The purpose of this instruction is to print a number of stack elements interpreted as booleans. If a stack element has the value 0, "false" is printed, otherwise "true" is printed.

"bool st[sr], 1" will print the top of the stack as a boolean, while "boolean st[st[sr]], 5" will print out the 5 stack elements up to the address contained in the stack top.

F.2.2 PrintChar.

"PrintChar" is similar to "PrintBoolean", but allows the printing of elements as characters.

F.2.3 PrintInteger.

"PrintInteger" is again similar to "PrintBoolean", allowing stack elements to be printed as integers.

F.2.4 PrintReal.

"PrintReal" allows the printing of two stack elements as a real value.

F.2.5 RegCommand.

The register command is designed to allow the examination and modification of the 4 registers "br", "ir", "sb" and "sr".

For example, "register sr:10" places the value "10" in the stack register "sr", while "reg ir: ir+1" allows the instruction register "ir" to be incremented.

F.2.6 StackCommand.

The stack command is similarly designed to allow the examination and modification of the stack elements. If no addresses are specified, the default stack elements to examine are the top 10.

For example, "stack" results in the top 10 stack elements being printed, "stack sr-5 : 11" places the value "11" in the stack element "st[sr-5]" while "stack sr-10, 5" results in the 5 stack elements "st[sr-14]···st[sr-10]" being printed out.

F.2.7 StepCommand.

The step command provides for a trace to made through the E- code instructions of a specification. By default this occurs from the present instruction register value, and consists of the execution of only one instruction. The start address and the number of instructions to execute can, however, also be specified.

For example, "step" results in the current instruction being executed. "step 10" results in the next 10 instructions being executed, while "step ir+6,

10" results in 10 instructions, starting at the instruction 6 words further on, being printed.

F.2.8 Quit.

This command allows the debugging session to be terminated.

In the next section the error conditions reported by the CLI are enumerated.

F.3 CLI Error Messages.

1. Illegal Character.
2. Comma Expected.
3. Extra Ignored.
4. Factor Expected.
5. Illegal Debugger Command.
6. Invalid Integer.
7. Left Square Bracket Expected.
8. Left Parenthesis Expected.
9. String Constant Too Long.
10. Right Square Bracket Expected.
11. Invalid Real.
12. Right Parenthesis Expected.
13. Invalid String.
14. Invalid Type in Expression..
15. Invalid Word.

E.2 Contents of Token File.

NewLine(1) *NewLine*(2) *NewLine*(3) *NewLine*(4) *NewLine*(5) *NewLine*(6)
NewLine(7) *NewLine*(8) *NewLine*(9) *NewLine*(10)
NewLine(11)
  Specification Identifier(25) *SystemProcess SemiColon*
  TimeScale Identifier(11) *SemiColon*
NewLine(12) *NewLine*(13) *NewLine*(14) *NewLine*(15) *NewLine*(16)
NewLine(17) *NewLine*(18) *NewLine*(19)
NewLine(20)
  Const
NewLine(21)
  Identifier(26) *Equal Integer*(0) *SemiColon*
NewLine(22)
  Identifier(27) *Equal Integer*(1) *SemiColon*
NewLine(23)
  Identifier(28) *Equal Integer*(2) *SemiColon*
NewLine(24)
NewLine(25)
  Type
NewLine(26)
  Identifier(29) *Equal Identifier*(26) *DoubleDot Identifier*(27)
  SemiColon
NewLine(27)
  Identifier(30) *Equal Record Identifier*(31) *Colon Identifier*(1)
  End SemiColon
NewLine(28)
  Identifier(32) *Equal Integer*(0) *DoubleDot Integer*(1) *SemiColon*
NewLine(29)
  Identifier(33) *Equal LeftParenthesis Identifier*(34) *Comma*
  Identifier(35) *RightParenthesis SemiColon*
NewLine(30)
  Identifier(36) *Equal*
NewLine(31)
  Record
NewLine(32)
  Identifier(37) *Colon Identifier*(33) *SemiColon*