THE DESIGN AND IMPLEMENTATION OF A FORTRAN-77 TO MODULA-2 TRANSLATOR

CENTRE FOR NEWFOUNDLAND STUDIES

TOTAL OF 10 PAGES ONLY
MAY BE XEROXED

(Without Author's Permission)

ERIC ROBERT MYHR, B.A., Dipl. C.S.

The Design and Implementation of a Fortran-77 to Modula-2 Translator

by

© Eric Robert Myhr, B.A., Dipl. C.S.

A thesis submitted to the School of Graduate Studies in partial fulfillment of the requirements for the degree of Master of Science

Department of Computer Science
Memorial University of Newfoundland
St. John's, Newfoundland
February 1990

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-65301-9

Abstract

A *source-to-source translator* is a program which translates programs written in a given high-level programming language into another high-level language. They provide a reliable means for the re-use, sharing, and development of software.

In this thesis, the design and implementation of a source-to-source translator which converts Fortran-77 programs into semantically 'equivalent' Modula-2 programs is described.

An *attribute grammar* is used to formally describe the translation. Attribute grammars are typically used in the specification of compilers and translators, and describe translation in a syntax-directed fashion.

The translator was generated from the attribute grammar using the GAG system, a translator/compiler writing system based on attribute grammars. Attributed parse trees are used for the intermediate representation of the syntax and semantics of Fortran programs during translation.

Keywords: *source-to-source translation, program transformation, attribute grammars, translator-writing systems, the GAG system, programming languages, Fortran-77, Modula-2.*

Acknowledgements

This thesis would not have been possible without the support and guidance of many individuals and institutions at Memorial University of Newfoundland.

Financial support was provided by the School of Graduate Studies through the University fellowship, by the Department of Computer Science through the provision of sessional lectureships, and by Dr. Wlodek Zuberek.

My thanks to Mrs. Jane Foltz for her careful review of my thesis, and for making financial support from the Department of Computer Science possible. I would also like to extend thanks to the technical support staff of the Department of Computer Science, who never tired of lending me a hand at any hour of the day or night.

Finally, I would like to extend very special thanks to my advisor, Dr. Wlodek Zuberek, for the thoughtful and patient guidance which he provided from the project's inception to its completion. From our numerous discussions evolved many of the ideas presented in this thesis.

Eric R. Myhr

Contents

List of Figures

Chapter 1

Introduction

Source-to-source translation is the process of translating between high level languages; it is a process by which programs written in a given high-level language (say S1) can be translated into another high-level language (say S2) using some kind of program transformations which preserve the meaning of (ie., the computation described by) the program written in S1. Ideally, the method employed should be sufficiently well-defined as to permit the automation of the translation. Automated translators are desirable for a number of reasons - they provide a reliable means to adapt existing software to new programming environments and to share programs with installations having limited source language facilities. They also allow software to be *re-used*; for instance, subprogram material written in obsolete languages can be converted and subsequently used in the construction of more powerful systems written in more *fashionable* languages. For the purposes of software prototyping, such a tool can be used to translate prototypes (written in a language suitable for prototyping) into a more production oriented language (as in [Dob 87]).

The practical aspect of the project described in this thesis consisted in the design and implementation of a Fortran-77 to Modula-2 translator[1]. While a fully general translator is beyond the scope of this thesis, the goal was to generate Modula-2 programs which are

[1] In the sequel, the term Fortran denotes Fortran-77 as defined in the ANSI standard [ANS 78], and Modula-2 denotes the version of the language described in Niklaus Wirth's "Programming in mboxModula-2" (third corrected edition, [Wir 85]) unless otherwise specified.

semantically equivalent to Fortran programs with as few *serious* restrictions imposed on the source programs as possible.

The underlying organization of the translation is modeled as a two- level mapping: Fortran programs are first described using some intermediate representation (IR); the second level maps the computation described in that intermediate representation onto the target language Modula-2. This raises a number of questions. First, how should the IR be designed? Secondly, how can the first level of mapping be described and effected? And thirdly, how to convert from intermediate representation of a program to its representation in Modula-2 ?

Clearly, the design of the IR is of central importance. The method used in this project is to represent the IR as an *attributed tree.* In such a scheme, attributes are used to express the meaning of various programming language constructs. Less abstractly, attributes (which annotate nodes of such a tree) should store enough information about the represented program to enable its *reconstruction* in the target language. This information (which is either explicitly represented by attributes of a tree or derivable from them) would need to include a description of declared objects (variables, types, constants, ...), actions (assignment, expression evaluation, binding), program structures (such as subroutine, function, and block data subprograms), control structures, data structures (arrays, common blocks), and relationships between entities (association of objects and structures; for instance *equivalenced* entities).

Having chosen an attributed tree representation, a number of design considerations come into play. Firstly, should a single (standard) IR having as fixed a structure as possible be used (as argued in [Tel 84]) or should a number of standard IRs be employed in the translation. The latter approach implies a need to describe transformations of attributed trees by either directly manipulating tree structures or by using attributes which are themselves ATs (the latter approach is taken in the attribute coupled grammar formalism described in [Gan 84], and in the higher order attribute grammar (HAG) formalism proposed in [Vog 89]). Such translations describe a multi-level mapping in which ATs are

successively transformed into some (possibly) standard or normalized form. The former approach (using a single IR) seems to be the most popular in the literature (this is the approach taken in [Alb 80], [Leo 87], [Boy 84], and [Sla 83]). Normally the translation proceeds by reworking the source program until it is in a form suitable for its representation in the standard IR. In [Sla 83] this reworking is kept fairly simple, and generation of the DIANA ([Goo 83] and [Tel 84]) tree is done directly on a statement-by-statement basis, while in [Boy 84] and especially in [Alb 80] the reworking is done in a number of more or less discrete steps (by, in the latter case, manipulating "non-standard" trees). In most cases, going from the IR to the target language code is done in a single step.

Another design consideration is whether the IR (when a single IR is used) should be target language oriented or source language oriented. In [Sla 83] and [Boy 84] the IR is clearly target language oriented. In [Alb 80] on the other hand, the IR is based on the syntax of Ada even for Ada-to-Pascal translation. In this case, however, a fairly close relationship between the IR and the target language is still maintained in virtue of the method employed in the translation ([Kri 84]) which is based on the definition of "compatible" sublanguages of Pascal and Ada. In the Ada-to-Pascal translation, Ada programs are first translated into their associated sublanguage, and thus the programs represented in the IR are already in a form which is (fairly) compatible with Pascal.

In the Fortran-77 to Modula-2 translator, a single (attributed parse tree) IR is used throughout the entire translation process. The IR is based on the syntax of Fortran-77. Attributes are used both to represent the semantics of Fortran-77 programs and to generate Modula-2 code, and no manipulation of the syntax tree is performed. The IR is source rather than target language oriented which complicates the task of generating Modula-2 code from the IR. But happily the two languages bear enough of a syntactic resemblance to minimize any subsequent difficulties. By the same token, since systems which utilize target language oriented IRs rely heavily on the *reworking* of source code, having a source language oriented IR significantly reduces the amount of reworking of Fortran programs required. Another advantage is that the attribute grammar formalism

can be used to completely describe the translation.

The attribute grammar (AG) formalism is a tool which is well suited for describing the mapping of source code (or more specifically, parse trees of source programs) onto attributed parse trees (APTs)[2]. APTs can capture both the syntactic as well as the semantic characteristics of source programs. AGs describe this mapping in a syntax directed fashion. A given AG is "built upon" the underlying context-free grammar of the source language, with attributes of APTs decorating parse trees of source language programs. Simply speaking, an AG is comprised of two components: a set of context-free rules which describe the syntax of the source language, and a set of *attribute evaluation rules* (also called *semantic rules*) which describe the values of attributes occurring at nodes of parse trees for source programs. Since AGs describe translations in terms of parse trees of the source language, it was convenient to base the IR on the syntax of Fortran-77.

Moreover, AGs are not particularly well suited for describing tree transformations. In order to do so, either the AG must be written such that some attribute(s) themselves are attributed trees(as in [Gan 84] and [Vog 89]), or some kind of extension to the AG formalism must be developed (as in [Mon 84], where AGs are combined with subtree replacement grammars). This consideration encouraged the use of a single IR in the translation.

The AG for translating Fortran-77 to Modula-2 can be conceptually divided into two parts: the first part (corresponding to semantic analysis phase of a traditional compiler) consists of evaluation of attributes which provide semantic information about the program being translated. The second part consists in the evaluation of attributes used for code generation. These two *parts* are implemented as (for the most part) distinct 'passes' over the parse tree.

[2]Originally conceived by Knuth ([Knu 68]) to describe the semantics of programming languages, AGs have become increasingly popular in compiler construction tasks, and a number of AG based compiler writing systems are currently available (among them is the GAG - Generator based on Attribute Grammars - system which is being used to implement the Fortran to Modula-2 translator).

1.1 Structure of the paper

The remainder of this thesis is structured in the following manner. Chapter 2 gives a formal definition of AGs along with an informal description of their semantics using an example. The GAG system is described briefly in chapter 3. In chapter 4 a comparison of Fortran-77 with Modula-2 is given, with particular emphasis on the features of the languages which play a significant role in the translation strategy. Chapter 5 contains a discussion on the strategies used to convert various Fortran constructs into Modula-2. Chapter 6 describes implementation details of the translator, including a discussion of the AG used to specify the translation, and some characteristics of the GAG generated translator. Chapter 7 gives some examples of input to, and output from, the translator. The final chapter contains concluding remarks, including a summary of the restrictions imposed on Fortran programs by the translator, some suggestions for the elimination of these restrictions, a description of the difficulties which arose during development of the translator, and discussion on the more successful results.

Chapter 2

Attribute Grammars

An *attribute grammar* (AG) is a context-free grammar (CF) along with a set of *attributes*, *semantic rules*, and *semantic conditions* in which a fixed number of attributes is associated with each nonterminal symbol in the CF grammar. The semantic rules are written so that each string generated by the CF grammar is associated with a value (normally given by attributes of the start symbol of the CF grammar). AGs are a useful tool for programming syntax-directed computations such as compiling and translating. In such applications the underlying CF grammar is the grammar of the source language, and the semantic rules are written so that the value of a string in the language is its translation into the target language.

The following section gives a formal description of attribute grammars, and is followed by an informal description of their semantics using an example.

2.1 Definition and notation

An AG is defined as a 5-tuple ([Kas 80], [DJL 88], [Yel 87]):

AG = (G,Attr,Val,Eval,Cond)

- G is a context-free grammar (which describes the syntax of the language); G = (N,T,P,Z) where N is the set of non-terminal symbols in G, T is the set of terminal

symbols in G, P is the set of productions in G, and $Z \epsilon N$ is the start symbol. Each production $p \epsilon P$ is written as:

$$p : X_0 ::= \alpha_0 X_1 \alpha_1 X_2 \ldots \alpha_{n_p - 1} X_{n_p} \alpha_{n_p}$$

where $X_i \epsilon N$ and $\alpha_i \epsilon T^*$ for $i = 0, \ldots, n_p$.

- Attr is the set of attributes. Attributes are associated with nonterminal symbols of G, ie., there is a mapping $N \longrightarrow 2^{Attr}$ which associates with each nonterminal $X \epsilon N$ a set of attributes (denoted Attr_X). Attr_X is partitioned into two disjoint sets, Inh_X and Syn_X (the *inherited* and *synthesized* attributes of X, respectively). Note that

$$\forall X \epsilon N, \ \text{Attr}_X = \text{Inh}_X \cup \text{Syn}_X,$$
$$\text{Attr} = \bigcup_{X \epsilon N} (\text{Syn}_X \cup \text{Inh}_X).$$

In a given production, an attribute a associated with symbol X_i is called an *attribute occurrence* and is denoted $X_i.a$. The set of all attribute occurrences in a production $p \epsilon P$ is

$$\text{Attr}_p = \{X_i.a \mid p = X_0 ::= \alpha_0 X_1 \alpha_1 X_2 \ldots \alpha_{n_p - 1} X_{n_p} \alpha_{n_p} \wedge a \epsilon \text{Attr}_{X_i} \wedge 0 \leq i \leq n_p\}.$$

- Val is the set of all attribute values.

- Eval is the set of *semantic rules* in AG. Semantic rules are associated with productions. A set of rules associated with production $p \epsilon P$ is denoted Eval_p. Each semantic rule in Eval_p defines the value of an attribute occurrence in p as a function of zero or more attribute occurrences in p. If $r \epsilon \text{Eval}_p$ is a semantic rule associated with the production p then r is of the form:

$$X_i.a \leftarrow f_{p,i,a}(X_{i_1}.a_1, X_{i_2}.a_2 \ldots, X_{i_k}.a_k),$$

$0 \leq i_j \leq n_p$ for $1 \leq j \leq k$, $0 \leq i \leq n_p$, and each $X_{i_j}.a_j \epsilon \text{Attr}_p$. For each $p \epsilon$P, Eval specifies exactly one semantic rule for each synthesized attribute of X_0 and one rule for each inherited attribute of X_i for $1 \leq i \leq n_p$ which ensures that the value associated with each attribute is uniquely determined in any context.

With any string in the language L(G) there is an associated derivation tree. Let s be a string in L(G) and let K_X be a node in its associated derivation tree which is associated with nonterminal $X \epsilon N$. An *attribute instance* $K_X.a$ is associated with each attribute $a \epsilon \text{Attr}_X$.

- Cond is a set of *semantic conditions* associated with productions $p \epsilon$P. A semantic condition associated with a production $p \epsilon$P, denoted Cond_p, is a boolean expression and has the following form:

$$g_p(X_{i_1}.a_1,\ X_{i_2}.a_2,\ \ldots,\ X_{i_k}.a_k)$$

where $0 \leq i_j \leq n_p$, $1 \leq j \leq k$, and each $X_{i_j}.a_j \epsilon \text{Attr}_p$. Semantic conditions are essentially conditions that must be satisfied by attribute values. A string $s \epsilon$L(G) is a string of the language L(AG) if and only if for all $p \epsilon$P the values of the attribute instances associated with *each* application of p in the derivation of s satisfy the condition Cond_p. The importance of semantic conditions lies in their ability to formally specify non-context-free aspects of a language, since strings in L(AG) are strings in L(G) which obey certain *context sensitive* constraints.

We have seen that for a given production $p \epsilon$P the semantic rules in Eval_p define the values of each attribute occurrence in Attr_p in terms of other attribute occurrences in Attr_p. This gives rise to the notion of *dependencies* between attribute occurrences. The *local* (or *direct*) *dependency relation* D_p defines the relation of local dependencies between

attribute occurrences in p:

$$D_p = \{< X_i.b, X_j.a > \mid X_j.a \leftarrow f_{p,j,a}(\ldots, X_i.b, \ldots) \in \mathrm{Eval}_p \wedge 0 \leq i \leq n_p \wedge 0 \leq j \leq n_p\}$$

The *local dependency graph* of production p is the graph of D_p:

$$\mathcal{G}_{D_p} = (\mathrm{Attr}_p, D_p).$$

Suppose $\mathcal{D}$ is a derivation tree for a string in G. The *compound dependency graph* $R_d(\mathcal{D})$ is the graph over the attribute instances in $\mathcal{D}$ which can be constructed by 'pasting together' the graphs $\mathcal{G}_{D_p}$ according to the applications of productions p in the derivation of the string.

An attribute grammar is *well formed* or *non-circular* if and only if for every derivation tree $\mathcal{D}$, $R_d(\mathcal{D})$ is *acyclic* ([DJL 88]).

An *attribute evaluator* for an $\mathcal{AG}$=(G,Attr,Val,Eval,Cond) is a program which, given any derivation tree $\mathcal{D}$ of a string in L(G) as input, computes the values of all the attribute instances in $\mathcal{D}$. An *evaluator generator* constructs (if possible) an evaluator for an attribute grammar given as input. That is, it is the job of an evaluator generator to determine a feasible *evaluation order* for the attribute instances of any derivation tree $\mathcal{D}$.

An evaluation order for the attribute instances in each derivation tree associated with an AG exists if and only if the AG is non-circular [DJL 88]. The problem of determining whether of not an AG is non-circular has been shown to be an exponentially hard problem, and consequently evaluator generators which construct evaluator generators for *any* well-formed AG are quite inefficient [Kas 82]. There are a number of special classes of AGs whose noncircular property is verifiable in polynomial time. The most notable of these (for our purposes) is the class of *ordered attribute grammars* (OAGs) which is the class of AGs accepted by the GAG system (discussed in the following chapter).

Attribute grammars can be used to formally specify a mapping from a source language to a target language. Provided the target language is sufficiently well defined, an AG used in this capacity may be said to formally specify the semantics of the source language.[1] The general idea behind such an AG is to associate with the start symbol Z of the CF grammar a synthesized attribute(s) which generates a target language program equivalent to a program in the source language. Evaluator generators, in practice, are compiler/translator generators; attribute evaluators are translators which, given a parse tree for any well formed program in the source language, generates target language code by evaluating the attribute instances in the tree.

The Fortran-77 to Modula-2 translator is described by an AG, which can be further processed by the GAG attribute evaluator generator to construct the translator.

2.2 An example

In this section, a fragment of an attribute grammar for the Fortran-77 PARAMETER statement is presented. The PARAMETER statement associates symbolic names with constant values in Fortran-77 programs. The production rules and their associated semantic rules and conditions are given below using ALADIN notation. ALADIN is an attribute grammar definition language which is the input language of the GAG system ([Kas 87]). Lines beginning with the keyword `RULE` contain context free production rules in which terminal symbols are enclosed within apostrophes (eg., `'PARAMETER'`). The semantic rules and conditions (the latter are preceded by the keyword `CONDITION`) associated with a production are given between the keywords `STATIC` and `END`.

An occurrence of an attribute `attr` associated with nonterminal symbol `X` is written as `X.attr`.

```
RULE 1:   param_stmt ::=  'PARAMETER' '(' const_defns ')'
STATIC    const_defns.env_in := param_stmt.env_in;
          param_stmt.env_out := const_defns.env_out
```

[1] This formalism is known as *translational semantics* [Pag 81].

```
END;

RULE 2:  const_defns  ::=  const_defns  ','  const_defn
STATIC   const_defns[2].env_in  := const_defns[1].env_in;
         const_defn.env_in      := const_defns[2].env_out;
         const_defns[1].env_out := const_defn.env_out
END;

RULE 3:  const_defns  ::=  const_defn
STATIC   const_defn.env_in   := const_defns.env_in;
         const_defns.env_out := const_defn.env_out
END;

RULE 4:  const_defn   ::=  name '='  arith_const_expr
STATIC   CONDITION undefined(name.symbol,const_defn.env_in);
         arith_const_expr.env_in := const_defn.env_in;
         const_defn.env_out      :=
           update_environment(const_defn.env_in,
              name.symbol,constant,arith_const_expr.code)
END;
```

In rule 2, `const_defns[1]` and `const_defns[2]` are formally the same symbol; the numerical suffixes enclosed in square brackets are used to distinguish between different occurrences of the same nonterminal symbol in a rule for specifying attribute occurrences in the semantic rules.

On the right hand side of each semantic rule is an *attribute expression*, which is made up of attribute occurrences, constants, and/or function invocations, and which specifies the value to be associated with the attribute occurrence appearing on the left hand side of the rule.

An attribute of a symbol may be thought of as a variable which associates with the symbol an aspect of its *meaning*. For instance, the attributes `env_in` and `env_out` are used to represent information about the environment of a program. Intuitively, the environment consists of a description of named entities in a program. The instances of `env_in` and `env_out` at a node labeled with symbol `param_stmt` represent the environment in which the parameter statement represented by the subtree rooted at that node occurs; `env_in`

represents the environment *prior* to encountering the statement, while **env_out** represents the environment *after* the statement has been encountered (ie., the *new* environment in which constants defined in the statement are described).

The attribute **code** of nonterminal **arith_const_expr** represents the code to be used for the constant expression in the corresponding constant definition in the target language code. The attribute **symbol** of the nonterminal **name** might be the index to the symbol table entry containing a description of the token associated with **name** in the source program[2].

To complete the example, the domains (ie., the sets of possible values) of the attributes should be described. The following (incomplete) type description written in ALADIN defines the type **environment** which is the domain of **env_in** and **env_out**:

```
TYPE  environment : LISTOF definition;
TYPE  definition  : UNION( const_defn, var_defn, ... );
TYPE  const_defn  : STRUCT( ident: SYMB,
                            code : STRING);
...
```

Here the domain **environment** is described as a list of definitions. The domain **definition** is described using the ALADIN discriminated union type, that is, a value in the domain **definition** can be a value in the domain **const_defn**, or **var_defn**, etc. The domain **const_defn** is a domain of pairs, the first element of which is a symbol and the second element of which is a string (**SYMB** and **STRING** are built-in ALADIN types for symbol and string values respectively).

The attribute **symbol** associated with symbol **name** is of type **SYMB**, while attribute **code** associated with attribute **arith_const_expr** is of type **STRING**.

An *attributed parse tree* (APT) is a derivation tree in which nodes labeled with nonterminal symbols $X \epsilon N$ contain *fields* corresponding to the attributes in Attr_X. Each of these fields corresponds to an attribute instance. An APT for the PARAMETER statement

```
PARAMETER (X = 5.2, Y = -43.2)
```

[2]A more complete example would include attributes for the type of the arithmetic expression, the type of the named constant (which would be included in the environment), and any type specification by IMPLICIT statements or type statements that may have preceded the PARAMETER statement.

is shown in figure 1. Attribute instances are denoted by the attribute names appearing immediately below the nodes (the values associated with them are not indicated).

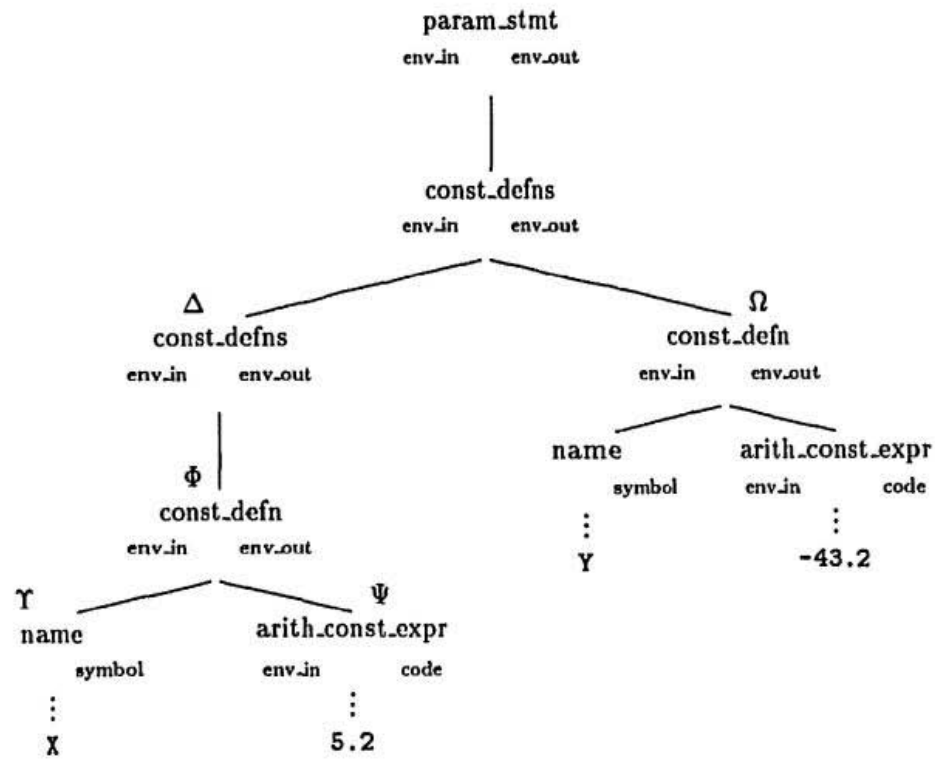

Figure 1. An APT for a Fortran **PARAMETER** *statement*

Consider the nodes labeled Φ, Ψ, and Υ[3] in the APT. The semantic rule

```
arith_const_expr.env_in := const_defn.env_in
```

specifies that the instances of `env_in` at Ψ and Φ will have the same value. The semantic rule for `const_defn.env_out` associated with production 4 indicates that the value of the occurrence of `env_out` at Φ will be the result of evaluating the invocation of

[3] Upper case Greek letters have no semantics in the represented tree, and are provided solely for reference in the text.

update_environment with arguments Φ.env_in, Υ.symbol, constant, and Ψ.code. The function (which would be defined elsewhere in the AG description) returns a new environment in which Υ.symbol (in this case, the symbol X) is defined as a constant associated with the code derived from the arithmetic expression (eg., '5.2'). The argument constant is a (user defined) scalar constant identifier which is supplied as argument to signal the function that the new entity is a constant.

The condition in rule 4 effectively stipulates that only previously undefined symbols may be defined in a given constant definition (a violation of this condition indicates that the source program is invalid). The function undefined is boolean typed, and must be defined elsewhere in the AG specification. For example, the constant definition represented by the subtree rooted at Φ is only valid if Υ.symbol (the symbol X) is not defined in the environment preceding its definition (Φ.env_in).

Recall that attributes associated with a given nonterminal symbol are classified as being either synthesized or inherited. In figure 1, inherited attribute instances are denoted on the left hand side of their associated node while synthesized attribute instances appear on the right. Intuitively, inherited attributes are used to pass information *down* the tree towards the leaves, while synthesized attributes propagate information *up* the tree towards the root. The attribute env_in in the example is an inherited attribute of the symbols param_stmt, const_defns, const_defn and arith_const_expr while env_out is a synthesized attribute of the same symbols. code and symbol are synthesized attributes of arith_const_expr and name, respectively. Note that the semantic rules associated with a given production specify a value for each synthesized attribute of the symbol on the left hand side of the production and for each inherited attribute of each nonterminal symbol appearing on its right hand side

Consider again the node Φ. The production applied at its parent is production 3. The value of env_in at Φ is obtained (*inherited*) from its parent; the semantic rule defining its value is associated with production 3. The value of env_out at Φ is expressed as a function of attribute values of its children, and the semantic rule defining its value is

associated with production 4 (the production applied at Φ). That is, the node inherits an environment (*via* **env_in**), it *synthesizes* a new environment using information derived from the subtree of which it is the root, and that new environment is passed back to (ie., synthesized by) its ancestors *via* **env_out**.

Now consider the node labeled with Ω in figure 1. The production applied at the parent of Ω is production 2. In this case, the environment inherited by Ω is the environment synthesized by its sibling Δ (which includes the definition of **X**).

Notice how values of **env_in** work their way down the tree, while values of **env_out** generally work their way up the tree. In this example it is not difficult to see that a single pass[4] (a kind of depth-first traversal) over the tree could be employed to evaluate its attribute instances. Had attribute instances at the internal nodes of the tree been (perhaps indirectly) defined in terms of the instance of **env_out** of the root, then a second pass would be required. In fact, a desirable feature of AGs lies in the ease with which such 'multi-pass' translation strategies can be specified.

[4]The term *pass* is used in a non-technical sense here. Actual strategies for attribute evaluation (eg., pass oriented, visit oriented) employed by evaluators depend on the type of evaluator generator used - see [Eng 84] for a description of evaluation strategies.

Chapter 3

The GAG system

The GAG system (Generator based on Attribute Grammars) is a translator writing system which generates translators for languages defined by attribute grammars ([Kas 82], [Ka2 87], [Kas 87], and [Hut 87] provide complete description, and [DJL 88] a brief description of GAG). The input to the system is an AG written in ALADIN. The AG must be an ordered attribute grammar (OAG). The output from the system is a translator written in Pascal, in which is embedded a user supplied scanner, a parser generated by PGS (Parser Generating System ([Gro 86]), and (optionally) some user supplied Pascal code.

3.1 ALADIN

ALADIN (A Language for Attributed DefINitions) is a strongly typed language. The types of all attributes and the symbols with which they are associated must be declared. The user may optionally specify whether an attribute is synthesized or inherited with respect to the symbols with which it is associated. If an attribute's class is specified, GAG requires that the class derived from the semantic rules agree with the declared class.

The predefined types in ALADIN are `INT` (integer values), `BOOL` (boolean values), `CHAR` (character values), `STRING` (sequences of `CHAR`), and `SYMB` (terminal symbols encoded in a symbol table). The user may define enumerated types (similar to those in Pascal),

subrange types, and structured types. The structured types are sets, structures (invariant record types), discriminated union type, and lists (whose elements are of fixed - though possibly structured - type). A few functions are provided for list manipulation (eg., `HEAD`, `TAIL`, `ELEM_IN_LIST`, etc.).

Attribute expressions (which appear in semantic rules and conditions) must obey ALADIN's type rules and must match the type of the attribute occurrences they define when used in semantic rules. A number of operators are provided for performing integer arithmetic, set operations, boolean operations, and list concatenation. No string operators are provided. With the exception of the set membership operator, all binary operators require identically typed operands.

Attribute expressions may contain function invocations. ALADIN allows the user to define functions (in ALADIN) which can be recursive. Such functions are *pure* functions; they do not have side effects. The concept of variables and control structures is absent from the language, so virtually all computations which might normally be implemented using iteration must be described recursively (even though GAG may implement recursive ALADIN functions using iteration in the Pascal code). A facility for defining *external* functions (written in Pascal) is also provided. External functions provide the only means of generating side effects (such as output) and directly accessing the translators' data structures (such as the symbol table). A common approach for generating output at translation time is to use boolean typed external functions in semantic conditions which have the side effect of producing output.

While ALADIN type rules are strict, values of some types may be *coerced* to values of other types using type conversion functions (such as when two values share the same base type). For instance, to add an element to a list, the element must be explicitly coerced to the list type before list concatenation can be specified. Also provided are type testing functions which are useful when discriminated union types are used.

Attribute expressions may be 'structured'. ALADIN provides a `CASE` expression (selection is done on the basis of the *type* of the case selector, not its value), a `LET` expression

(as in LISP), and an `IF THEN ELSE` expression. Structured expressions can be nested to arbitrary depth.

Productions are written in a restricted EBNF form, in which alternation is forbidden and neither repetitive nor optional clauses can be nested. Semantic rules are either 'normal rules' (such as those in the example of the previous chapter), semantic conditions, or transfer rules. Semantic conditions may optionally include a message which is output if the condition fails at translation time. Transfer rules are abbreviations of one or more *copy rules*. A copy rule is a rule of the form $X.a_i := Y.a_j$ in which an attribute value is transfered without modification from one attribute instance to another. The semantic rules associated with production 3 in the example of the preceding chapter are of this type and could have been expressed using the following ALADIN transfer rule:

```
TRANSFER env_in, env_out;
```

ALADIN also allows *non-local* attribute occurrences to be referenced in semantic expressions. So-called *outer attributes* (attributes of symbols from which the left hand side of a production are derived) can be referenced using the `INCLUDING` clause, while *inner attributes* (attributes of symbols derived from symbols on the right hand side of a production) can be referenced using the `CONSTITUENTS` clause.

3.2 GAG processing

The processing of ALADIN input is done in a number of passes, some of which are optional. These are summarized below and appear in the order in which they are executed by GAG.

Syntactic and semantic analysis. In the first pass, the ALADIN text is scanned and checked for syntactic correctness. Semantic checking is primarily concerned with verifying that the ALADIN type rules are obeyed in the semantic rules, conditions, and ALADIN functions.

Expansion. In this pass, transfer rules are rewritten as a set of equivalent semantic rules. In addition, attributes and semantic rules are introduced to perform the transport of non-local (inner and outer) attribute values which appear in `INCLUDING` and `CONSTITUENTS` clauses.

Chain elimination. APTs are represented by the translator using records with pointers, and some optimization is performed to reduce the size of trees generated at translation time. Among them are the elimination of *chain rules* and 'useless' terminal symbols from the AG. Chain rules are those in which only a single nonterminal symbol (along with zero or more terminal symbols) appears on the right hand side of a production and whose associated semantic rules are transfer rules only (an example is rule 3 of the example in the previous chapter). The elimination of chain rules performed by this pass effects only the tree construction routines generated by GAG (and not the PGS generated parser).

Dependency analysis. The next pass is concerned with the analysis of attribute dependencies in the grammar. GAG accepts only ordered attribute grammars. OAGs are described formally in [Kas 80] and only a few of their properties are given here.

OAGs constitute a large subcla. of non-circular AGs. An AG is an OAG if "for each symbol a partial order over the associated attributes can be given, such that in any context of the symbol the attributes are evaluable in an order which includes that partial order" ([Kas 80]). OAGs form a sufficiently large class for specification of programming languages. The complexity of the problem of determining whether an AG is an OAG is polynomial in the size of the AG.

During this pass a complete analysis of attribute dependencies based on the construction of dependency graphs is performed, and 'visit sequences' based on the ordering property of the AG are generated. Visit sequences are tree-walking rules which control attribute evaluation in the generated translator. The algorithm for the generation of visit sequences is presented in [Kas 80]. GAG allows the user to specify that alternative visit strategies (eg., *pass oriented*) be employed.

Optimization. In this pass the system determines whether some optimization of the storage of attribute values can be performed. The 'life-time' of attributes is examined to determine whether their values can be stored in global variables or global stacks. Only attributes which cannot be stored in this manner are actually stored in the fields of APT nodes.

Translation of visit sequences. In this pass, a space optimized representation of the visit sequences generated by the analysis of dependencies pass is generated. Visit sequences are stored in a table in the generated translator at translation time, and the output from this pass is a file which is used to initialize the table. Identical visit sequences may be stored in the same table entry.

Syntax translation. This pass creates a description of the CF grammar in the AG in a form suitable for processing by PGS, in which connection points are specified providing the interface between the generated parser and the translator. A file is also output which is read by the generated compiler to initialize its lexical analyzer.

Translation of definitions and actions. This pass is responsible for generating Pascal code which implements global definitions, attribute evaluation algorithms, and supporting procedures for the translator.

Protocol generation. This pass produces a listing of the ALADIN text in which all error messages, warnings, and informationals are merged. The user may also specify that additional information be provided by the various passes including: a listing of attribute dependencies, a listing of eliminated chain rules, a cross-reference listing, a listing of generated visit sequences, and a listing of information about the result of attribute optimization.

3.3 Structure of the generated translator

A GAG-generated translator is comprised of five components: a scanner, parser, tree constructor, attribute evaluator, and external definitions. We describe these briefly here.

The scanner is supplied by the user. GAG provides a *sample* scanner (capable of scanning any Pascal token) which the user must modify to scan the tokens of the source language. Additional *types* of tokens can be introduced, although a good deal of care must be taken when doing so as the interface between scanner and parser is not particularly obvious.

The parser is constructed (after the GAG passes described in the previous section) by PGS from the context free rules extracted from the ALADIN text by GAG. PGS is an LALR(1) parser generator, and the input grammar *must* have the LALR(1) property.

Pascal procedures for tree construction are generated by GAG. In the input supplied to PGS, the code for the invocation of node-building procedures is associated with the context free rules.

The external definitions are user defined Pascal definitions. In addition to ALADIN externals, the user can specify that definitions and declarations of constants, types, variables, external files, functions, and procedures be defined globally in the GAG constructed translator. Moreover, the user can supply Pascal statements to be executed either prior to or after execution of the GAG generated statements in the body of the main program.

These components, along with the routines and tables for attribute evaluation, are merged together by a program preprocessor (PROPP) into a single Pascal program. The input to the translator consists of four files which contain: the source code to be translated, the visit sequences, symbol table values, and a parser table (generated by PGS).

Chapter 4

A comparison of Fortran-77 with Modula-2

In this chapter some of the important differences between Fortran and Modula-2 are discussed. Other differences are addressed in subsequent chapters. Attention here is restricted to those features of the languages which have a direct bearing on the problem of translating Fortran-77 into Modula-2.

4.1 Program structures

4.1.1 Modules in Modula-2

In Modula-2, the primary program structure is the *module*[1]. In general, a Modula-2 program consists of a *main program module* and a number of *subsidiary* modules. Modules are syntactically similar to parameterless procedure definitions in that they contain definitions (of constants, types, procedures, functions, etc.), declarations (of variables, arrays, etc.), and bodies (called *module bodies*) consisting of executable statements. A desirable feature of module structured languages is the ability to group together related objects.

[1] A detailed description of the module concept is beyond the scope of this report, and the interested reader is referred to [Gle 84] for a short but readable description of the concept.

For example, one subsidiary module might be used to contain procedures and variables for performing input and output, while another might contain procedures for performing operations on strings (unlike Fortran, Modula-2 does not provide built-in string operators).

A key semantic difference between modules and procedures lies in the *visibility* of their locally declared objects. An object declared in a module can be made visible and accessible outside of the module if the module *exports* that object. Conversely, objects declared outside a module can be made visible within that module if the module *imports* that object (which must be exported by some module).

Another key difference concerns the *existence* of locally declared objects. While objects declared in a procedure only exist during the execution of the procedure, objects declared in a module exist throughout execution of the program, whether or not the module containing their declaration is the main program module.

While modules (like procedures) may be nested, they can also be *separately compiled* and stored in compiled form. Once a main program is compiled, it is linked with the precompiled modules from which it imports objects. When a separately compiled module is imported in more than one place, Modula-2 defines that only one instance of the module exists at a time. Separately compiled modules consist of two *compilation units*: a *definition module* and an *implementation module.*

The definition module contains declarations of the module's exported objects, such as constants, types, variables, and procedures[2]. Implementation modules contain the code that implements the objects defined in the definition module (when necessary). Implementation modules may additionally contain locally declared objects which aid in that implementation, but such objects cannot be exported. Implementation modules also contain module bodies, and the statements contained therein are executed *prior* to the execution of the main program (module bodies are used to assign initial values to objects declared in the modules; the main program module body is the body of the main program).

[2]Procedure declarations in definition modules consist of a procedure header only; their complete definitions are provided in implementation modules.

4.1.2 Fortran program units

An *executable* Fortran program is made up of one or more *program units*: a *main program* unit, and zero or more *subprogram units* which are either *subroutines*, *external functions*, or *block data subprograms*. Subroutines and external functions are called *external procedures*: an external procedure may be invoked from the main program unit and other external procedures.

In general, subroutines and external functions are similar to mboxModula-2's procedures and functions, respectively. Only block data subprograms have no direct counterpart in Modula-2, although their role in an executable Fortran program can be simulated in Modula-2 by a module which declares, exports, and (if specified) gives initial values to objects corresponding to common block structures in the Fortran program.

The translation of Fortran to mboxModula-2 uses a distinct module for each program unit in an executable Fortran program. The main program unit is translated into the main program module, and external procedures are translated into distinct separately compiled modules. A subroutine, for instance, would be converted into a module which defines and exports the procedure corresponding to the subroutine. If, moreover, that subroutine contained any invocations of other external procedures, then the module would import the definitions of the called procedures from the modules containing *their* definitions.

In addition to the modules corresponding to program units of a given Fortran program, a few modules are constructed which provide tools for performing complex arithmetic, exponentiation, input and output, and storage for common blocks.

It should be noted that it would be possible to convert an executable Fortran program into a Modula-2 program consisting of a single module (in which modules are nested). Modula-2's facility of *separate compilation* of modules does, however, bear a *resemblance* to independent compilation of Fortran program units as it is typically implemented[3].

[3]Separate compilation in Modula-2 and independent compilation in Fortran are *not* however identical! ([Wir 83], pg. 80). In order to compile a module which imports objects from another (separately compiled) module, the Modula-2 compiler needs a description of the imported objects - this description is provided the definition module of the imported modules).

The decision to break the target program down into separately compiled modules which have a more or less one-to-one correspondence with the program units constituting the Fortran program was made to exploit this similarity and keep the modules of the translated programs as *independent* as possible.

4.2 Parameter passing mechanisms

In Fortran, arguments to both function and subroutine subprograms are passed using a mixture of pass-by-reference and pass-by-value. In the terminology of the Fortran standard, dummy arguments are *associated* with actual arguments during subprogram execution. The nature of this association for a particular actual argument/dummy argument pair depends on whether the actual argument has an *l-value* ([Ten 81]) or not. If the actual argument has an l-value (such as when the actual argument is the name of a variable, array, or array element) pass by reference is used. When the actual argument is an expression which does not have an l-value, the corresponding dummy argument becomes associated with the r-value of the expression supplied as actual argument. Unlike a dummy argument which has become associated with an l-value, a dummy argument which becomes associated with an r-value may not be defined or redefined.

Modula-2 supports two kinds of formal parameters: *variable* and *value* parameters. Each corresponds to a different parameter passing mechanism, the former to pass-by-reference, and the latter to pass-by-value. The formal parameter list of a Modula-2 procedure explicitly indicates whether each parameter is a variable or value parameter[4].

An actual parameter corresponding to a formal variable parameter must have an l-value. An actual parameter corresponding to a formal value parameter must have an r-value, and the formal parameter is considered as a local variable in the subprogram which is initialized to the r-value supplied as argument in the invocation. Once this initialization

[4]Formal parameter lists in Modula-2 are syntactically similar to those of Pascal: if a formal parameter declaration is preceded by the keyword VAR then it is a variable parameter - its absence indicates that the formal parameter is a value parameter.

has been performed, there is no further association between actual and formal parameter, and the value of the formal parameter may be freely modified within the subprogram without any effect on the actual parameter. In this sense, there is no counterpart in Fortran to value parameters in Modula-2.

Thus in Modula-2, the parameter passing mechanism used for a particular parameter is determined not by the nature of the actual parameter used in its invocation (as in Fortran) but by the declaration of the formal parameter. Moreover, and perhaps more ominously, while in Modula-2 the parameter passing mechanism is *fixed* for a given formal parameter, in Fortran the argument passing mechanism used for a given dummy argument may differ from one invocation to the next!

Clearly, then, one cannot hope to precisely preserve the relation between actual and dummy arguments when translating between the two languages. The strategy employed in the translator's treatment of argument passing is discussed in section 5.6. The general idea is to pass arguments uniformly by reference. When an actual argument is an expression, an auxiliary variable is introduced to store the value of the expression immediately prior to invocation, and that variable is used as actual argument in place of the expression. This approach has the additional advantage of permitting subprograms to be translated independently.

4.3 Passing arrays as argument

Another area of concern regarding the passing of arguments involves the correspondence of the types of actual and dummy arguments. In both Fortran and Modula-2, the type and number of actual and dummy arguments must in some sense *match* in the invocation and definition of a subprogram. But the sense in which array arguments must match is dramatically different in the two languages.

Modula-2 is rather strict, generally requiring that the types match *exactly*, and that

the type identifier used in the declaration of both have the exact same defining occurrence[5] (this is known as *name equivalence*). In Fortran if an actual argument is an array name, then the number and size of the dimensions of the actual argument array may differ from those specified in the declaration of the corresponding dummy argument array, provided that the size of the latter does not exceed that of the former.

Another discrepancy lies in the treatment of array elements as actual arguments. In Fortran, when an array element is used as actual argument the corresponding dummy argument can be either the same type as the array element *or* an array with the same element type as the actual argument array (the first element of which becomes associated with the array element supplied in the invocation). The latter possibility effectively allows *parts* of arrays to be passed to subprograms. In Modula-2, if an actual argument is an array element, the corresponding formal parameter must be of the same type as the array element supplied as argument. Consequently, only entire arrays or single array elements may be passed to Modula-2 procedures.

The only flexibility provided by Modula-2 in the passing of arrays is the *open array parameter*. Open array parameters are array formal parameters whose element type is declared but whose index type is not specified. Actual parameters corresponding to open array parameters can be any array of the same element type. The index type of an open array parameter is a subrange of CARDINAL[6] whose lower bound is zero. The upper bound depends on the length of the actual argument array. Modula-2 provides the standard function HIGH which returns the upper bound of an open array parameter when supplied with its name as argument.

An example of the use of an open array parameter is the following:

```
PROCEDURE ZeroElements (VAR A: ARRAY OF INTEGER) : INTEGER;
(* assigns value zero to each element in A *)
VAR i: INTEGER;
BEGIN
```

[5]The only exceptions to this rule occur in passing procedures (a procedure's type is determined implicitly in its definition) and *open array parameters*.

[6]The domain of CARDINAL type in Modula-2 is a set of nonnegative integers.

```
  FOR i := 0 TO HIGH(A) DO
    A[i] := 0
  END;
  RETURN
END ZeroElements;
```

Any array of integers, regardless of its index type, can be supplied as argument to the above procedure[7]. The following are valid invocations of the procedure:

```
(* declarations *)
VAR arr1 : ARRAY [-99 .. 100] OF INTEGER;
    arr2 : ARRAY [7 .. 8] OF INTEGER;
    arr3 : ARRAY [1..10, 1..10] OF INTEGER;
    ...
BEGIN
  ZeroElements(arr1);
  ZeroElements(arr2);
  ZeroElements(arr3[5]);
   ...
```

Notice that when `ZeroElements` is being executed with `arr1` as argument the formal parameter `A` is implicitly of the following type:

```
ARRAY [0..200] OF INTEGER
```

while in the third invocation, the type of `A` is:

```
ARRAY [0..9] OF INTEGER
```

Thus, while an open array parameter permits arrays of varying size to be supplied as actual argument, it does not allow the size of a formal parameter array to differ from that of the actual argument array during a particular invocation of the procedure. That is, an open array parameter is always considered to have the exact same number of elements as the actual argument[8]: this effectively prevents Modula-2 programs from passing parts of arrays to procedures. Moreover, open array parameters always have the same number of dimensions as the actual argument[9].

---

[7] Note that `A` is a variable parameter - open array parameters can also be value parameters.

[8] This also distinguishes open array parameters from *adjustable arrays* ([ANS 78], pg. 5-7) in Fortran.

[9] This distinguishes open array parameters from *assumed size arrays* in Fortran ([ANS 78], pg. 5-7).

These considerations highlight the more liberal concept of storage association found in Fortran with which structurally distinct entities can share contiguous storage locations. While in Modula-2 structurally different entities can share the same memory location(s) only through the use of variant record structures[10], in Fortran such association can be specified in the passing of arrays to subprograms, EQUIVALENCE statements, and multiple definitions of common blocks (see following chapter).

4.4 Passing procedures as argument

Both Fortran and Modula-2 support passing procedures and functions as argument. In Fortran, a name which appears in an EXTERNAL statement represents an external procedure and may be passed as argument provided the corresponding dummy argument name is also declared to be EXTERNAL in the invoked subprogram. That dummy argument can then be invoked provided the invocation is consistent with the definition of the actual argument subprogram (ie., the type and number of arguments must match and the context of the invocation must be consistent with the type - function or subroutine - of the represented subprogram). Moreover, such a dummy argument may be passed as argument to other subprograms.

In Modula-2, procedures can appear as actual arguments provided that the scope of the procedure name includes the place of invocation (the scope of a procedure name imported by a module includes the whole module unless another entity with the same name is declared in a procedure or module nested within it). A formal parameter which is used to represent a procedure must be declared with a procedure type.

Modula-2 provides a means of defining *procedure types* in which the type of parameters is described (including the parameter passing mechanism - by value or by reference) along with, in the case of function procedures, the type of the value returned. A procedure type

[10]Note that the association of differently dimensioned arrays in Fortran cannot, in general, be expressed in Modula-2 solely through the use of variant records: this is a consequence of the different storage allocation schemes used by the two languages for multidimensional arrays.

IntToBool, the domain of which is the set of function procedures which have a single (value) parameter of type INTEGER and which return a value of type BOOLEAN, is defined as follows:

```
TYPE  IntToBool = PROCEDURE (INTEGER) : BOOLEAN;
```

An example of a function which is *implicitly* of type IntToBool is the following:

```
PROCEDURE Less_than_five (n:INTEGER) : BOOLEAN;
(* returns true iff n is less than 5 *)
BEGIN
  RETURN n < 5
END Less_than_five;
```

Less_than_five (and any other boolean function which has a single value parameter of type integer) can be supplied as argument to any procedure which has a formal parameter of type IntToBool, such as the following:

```
PROCEDURE FindFalse (f: IntToBool): INTEGER;
(* returns the smallest non-negative integer n for which
 f(n) is false. Doesn't terminate if f(n) is never false *)
  VAR n:INTEGER;
BEGIN
  n := 0;
  WHILE f(n) # 0 DO  INC(n)  END;
  RETURN n
END FindFalse;
```

Note that the formal parameter f *could* be passed as argument from within FindFalse to a procedure having a formal parameter of type IntToBool.

A program containing the above definitions could invoke FindFalse with argument Less_than_five as in:

```
Some_Int_Var := FindFalse(Fact_less_than_cube);
```

4.5 Input and Output

In Fortran, input and output is record oriented. Input and output statements access records on files, and access can be either sequential or direct. Format descriptions or

implicit formatting is used in conjunction with input and output statements to specify the form and representation of records stored on files.

Fortran provides a number of statements for input and output. The READ, WRITE, and PRINT statements cause transfer of data between main memory and files[11]. A number of *file-positioning* statements (BACKSPACE, ENDFILE, and REWIND) can be used to control the position of the read/write device connected to a file. Also provided are three *auxiliary* input and output statements, the OPEN, CLOSE, and INQUIRE statements, which control the connection between a file and an executable program.

Modula-2 provides neither statements nor built-in procedures for performing input and output. Programmers must either import procedures from library modules supplied by a Modula-2 system or write their own. Presently, no standard library of I/O modules exist, although [Wir 85] describes a collection of four I/O modules: `Terminal` (for simple character oriented I/O with the users terminal), `InOut` (for sequential I/O of integer, cardinal, boolean, character and string values), `RealInOut` (for I/O of REAL values), and `FileSystem` (for more general and lower-level file handling operations such as opening, closing, creating, deleting, and renaming files, binary I/O, and direct access to files).

The lack of a standard I/O library necessarily renders any strategy for the translation of Fortran I/O non-standard. Although many Modula-2 systems provide the modules suggested by Wirth, they are *not* part of the Modula-2 language, and systems are not constrained to supply them. TopSpeed, for instance, provides only two modules for I/O: `IO` and `FIO`. `IO` is the library of procedures which provide I/O with the standard input and output devices (ie., the keyboard and screen). `FIO` provides file handling and file input/output. Files are sequential, but a procedure is supplied for setting a file's position which can be used to implement direct access to elements of a file.

Modula-2 is not record-oriented. While a single Fortran READ, WRITE, or PRINT statement specifies the transfer of a record (which can represent several values, one in each

[11]The READ and WRITE statements can also be used to transfer data between variables to change its representation by using an *internal buffer* instead of a file during transfer.

of its fields) between main memory and some I/O device, a given Modula-2 I/O procedure reads or writes only a single value of a particular type. Moreover, Modula-2 is primarily stream-oriented. With few exceptions, values in input files must be delimited by some character from a set of separators (in the same vein as list directed input in Fortran).

Chapter 5

Strategy of the Translation

5.1 Mapping of Types

In Fortran, values of type INTEGER, REAL, and LOGICAL all occupy one *numeric storage unit* in a storage sequence, DOUBLE PRECISION and COMPLEX values occupy two numeric storage units, and a CHARACTER datum has one *character storage unit* in a storage sequence for each character in the string. This mapping allows any *association* (sharing of storage units) of entities of different types to be well defined regardless of the machine on which Fortran programs are compiled *with standard conforming compilers.* Wirth's "Programming in Modula-2" (3rd corrected edition) does not specify such a mapping; only typical storage conventions are given. In the translation, it is desirable to preserve this relationship between the storage requirements of the various types. Since the target environment of the translator is TopSpeed Modula-2 system [TSD 88], the mapping of types is geared to that system. In TopSpeed Modula-2, the basic unit of storage is an 8-bit byte. The storage requirements for the types of interest are given below.

- INTEGER 2 bytes
- LONGINT 4 bytes
- REAL 4 bytes

- LONGREAL 8 bytes
- BOOLEAN 1 byte
- CHAR 1 byte
- LONGWORD 4 bytes

REAL values in Fortran are represented as REAL values in Modula-2. For Fortran INTEGER values to occupy the same amount of storage as REAL values, INTEGER type in Fortran is translated into LONGINT type in Modula-2 . A basic character unit of storage in a Fortran program is mapped onto type CHAR in Modula-2. DOUBLE PRECISION values in Fortran are mapped onto LONGREAL values in Modula-2. Other mappings are described in the following sections.

5.1.1 LOGICAL data

LOGICAL type in Fortran is mapped onto BOOLEAN type in Modula-2. However, LOGICAL entities[1] in Fortran are not translated into BOOLEAN entities in Modula-2 since only one byte of storage is used to hold boolean values in Modula-2. Consequently, a straightforward mapping of LOGICAL entities onto BOOLEAN entities would cause problems whenever logical and numeric type entities are associated. Thus, LOGICAL entities are translated into entities of TOP-SPEED Modula-2 type LONGWORD (which has the same storage requirements as REAL and LONGINT). LONGWORD type has properties similar to the standard Modula-2 type WORD. It can be used to store values of any type (provided some storage requirements are met).

While the only operation that can be performed on variables of type WORD and LONGWORD is assignment of WORD or LONGWORD values, respectively, using type transfer functions this restriction can be overcome, and values stored in LONGWORD

[1]The term *entity* is used throughout to refer to any named object which can be typed, such as variables, arrays, functions and constants.

typed entities can be *interpreted* as BOOLEAN values thereby permitting boolean operations to be performed on them.

For example, a variable declaration in a Fortran program such as

```
LOGICAL FLAG, FOUND, BITS(100)
```

could be translated into the following:

```
FLAG: LONGWORD;
FOUND: LONGWORD;
BITS: ARRAY [1..100] OF LONGWORD;
```

In the Modula-2 code, we could interpret the value of the variables above by supplying them as argument to the type transfer function BOOLEAN. For example, the invocations `BOOLEAN(FLAG)` and `BOOLEAN(BITS[33])` return the boolean values corresponding to the contents of `FLAG` and `BITS[33]` respectively. Effectively this allows us to perform logical operations on variables of type LONGWORD. Naturally, care must be taken to store BOOLEAN values in such variables consistently. This is achieved by transferring BOOLEAN values into type LONGWORD before storing them; whenever an assignment is made to a "logical" variable, the expression is converted to type LONGWORD. For instance, `FLAG` would store the value `TRUE` after the execution of any of the following assignments:

```
FLAG := LONGWORD(TRUE);
FLAG := LONGWORD((88 < 108) & (43 >= -4));
```

In order to make the Modula-2 program more readable, the translator defines the following type:

```
TYPE  Logical = LONGWORD;
```

This makes available a type transfer function called `Logical` which is equivalent to the function LONGWORD.

Another example follows. Suppose a Fortran program contains the following statements:

```
LOGICAL FOUND, LIMIT
FOUND = .FALSE.
LIMIT = .FALSE.
IF (.NOT. FOUND .AND. .NOT. LIMIT) THEN
  ...
```

The translation into Modula-2 would be the following:

```
(* declarations *)
  FOUND : Logical;
  LIMIT : Logical;
(* statements *)
  FOUND := Logical(FALSE);
  LIMIT := Logical(FALSE);
  IF NOT BOOLEAN(FOUND) & NOT BOOLEAN(LIMIT) THEN
    ...
```

5.1.2 COMPLEX data

COMPLEX entities in Fortran are translated into record structures with two fields of type REAL in Modula-2 (the first field corresponding to real part of the complex value, and the second corresponding to the imaginary part). The type `Complex` is provided by the translator, and is defined in Modula-2 programs as follows:

```
TYPE   Complex = RECORD
                   rpart:      REAL;
                   ipart: REAL
                 END;
```

Complex literals are translated into invocations of a function called `complex`, which takes two real values as argument and returns a record of type `Complex`[2]. Thus, a Fortran complex literal such as `(2.443,5.3)` is translated into the function invocation `complex(2.443,5.3)`. A value of any of the other numeric types can be converted to type complex (if required) by supplying them as argument to the same function. For

[2]Since the appearance of the third edition Wirth has stated that Modula-2 systems are free to allow functions to return values of any type (although the syntax of Modula-2 prohibits using the value returned by a function as a designator) ([Kin 88], pg. 127). TopSpeed Modula-2 allows structured types to be returned by functions.

instance, the real and integer values 8.33 and 1003 could be converted by the calls `complex(8.33,0.0)` and `complex(REAL(1003) ,0.0)`, respectively.

A consequence of this decision is that complex literals may not appear in constant expressions (since neither user defined nor library functions can appear in constant expressions in Modula-2) and consequently the translation of complex constants in constant expressions cannot be handled by the translator.

5.1.3 Character strings

Modula-2 represents strings as arrays whose elements are CHAR values and whose lower index bound is zero[3]. For example, the implicit type of the string literal `'Bad Day'` is:

```
ARRAY [0..6] OF CHAR
```

The translator uses the same convention for representing character string variables. For instance, if a program unit contains the following declaration:

```
CHARACTER NAME1*10,NAME2*20
```

then the corresponding declarations in the Modula-2 code is:

```
NAME1: ARRAY [0..9] OF CHAR;
NAME2: ARRAY [0..19] OF CHAR;
```

While it may seem more natural to use 1 as the lower bound (the Fortran convention for representing strings), the uniformity of string representation resulting from the use of zero-indexing for string variables ultimately eases the task of implementing string operations (which are not pre-defined in Modula-2[4]). The greatest advantage is in the ease with which string arguments of varying length can be passed to procedures using open array parameters. The tradeoff is that indices used in substring expressions and invocations of some string handling functions need to be offset by one.

[3] Wirth does not make it clear whether this is a language requirement or simply a convention [Kin 88].

[4] Most Modula-2 systems provide string handling function libraries, which typically assume a lower bound of zero for string arguments.

5.2 Constants

Integer constants in Fortran are syntactically the same as integer constants in Modula-2. The only difference between their treatment is that in Fortran blanks between digits have no effect on the value of the constant. This disparity can be handled during scanning of Fortran programs, although the current translator simply assumes that no spaces occur within numeric constants.

String constants in Fortran are delimited by apostrophes ('). An apostrophe can be quoted by having two in succession. In Modula-2, either the double quote character (") or the apostrophe character can be used to delimit a string literal, but the delimiting character may not appear in the string. So in the translation, strings are delimited by double quote characters (which are not in the standard Fortran-77 character set), and quoted apostrophes are replaced by a single apostrophe character.

There are a few differences between real constants in Fortran and Modula-2. In Modula-2, a real constant may not begin with a decimal point. In the translation, a leading zero is prefixed to any Fortran real literal starting with a decimal point. Double precision values in Fortran are basic real constants followed by a double precision exponent (eg. 12.5D-21). In Modula-2, the representation of LONGREAL literals varies from system to system. In TopSpeed, the syntax of both REAL and LONGREAL literals is the same (that is, a REAL literal can be used as either a REAL or a LONGREAL value - its context determines its interpretation). Such literals can also be used in expressions with operands of either type. Double precision literals are thus simply translated into Modula-2 real literals (the D's are replaced by E's). Real or double precision literals in Fortran with an exponent part need not contain a decimal point while a decimal point must be present in Modula-2 real literals. Consequently, such Fortran constants are modified to contain a decimal point before the exponent part.

5.3 Expressions

The greatest difference between expressions in Fortran-77 and Modula-2 is that while the former permits *mixed mode* arithmetic expressions, the latter does not and values combined in an expression must be (pairwise) of the same type. Thus, when numeric values of varying types appear in a Fortran arithmetic expression, type transfer functions are used to convert operands to the same type. This conversion is done in accordance with the Fortran rules for the type and interpretation of the results of arithmetic expressions given in [ANS 78] (Tables 3 and 4). For example, the Fortran expression `A * M + DP` where `A` is type REAL, `M` is type INTEGER, and `DP` is type DOUBLE PRECISION would be translated into the Modula-2 expression `LONGREAL(A * REAL(M)) + DP`.

To deal with arithmetic operations on complex numbers, the translator provides a number of functions that perform arithmetic operations on complex arguments. For example, a function called `addcomplex` is provided which takes two `Complex` records as argument and returns a record value corresponding to the sum of the arguments. The other functions supplied are `subcomplex`, `multcomplex`, `divcomplex`, and `raisecomplex` which replace the subtraction, multiplication, division, and exponentiation (raising a complex number to an integer power) operators in Fortran expressions using complex operands. When a complex value is combined with a numeric value of another numeric type, the function `complex` (described above) is used to convert the non-complex value to its corresponding complex representation. For instance, the Fortran expression

```
(8.6,6) * 72 + (1.9,3.4)
```

is translated as follows[5]:

```
addcomplex(multcomplex(complex(8.6,REAL(6)),
                       complex(REAL(72),0.0)),
           complex(1.9,3.4))
```

[5] Because of Modula-2's strict type checking, integer components of complex values must be converted to type REAL.

Operators in Fortran and Modula-2 do not share the same precedence relations; the logical operators NOT, AND (also denoted &), and OR in Modula-2 have the same precedence as arithmetic operators for unary minus, multiplication and division, and addition and subtraction respectively (as in PASCAL). Thus some additional parentheses must sometimes be added to expressions (in fact, it suffices to parenthesize relational expressions in logical expressions so that no attempt is made to perform logical operations before arithmetic operations).

The Fortran operators .EQV. and .NEQV. are absent in Modula-2 but the same operations can be specified using relational operators for equality ("=") and inequality ("<>" or "#"), respectively, with logical operands.

No exponentiation operator is provided in Modula-2, and consequently the translator converts exponentiation operations into function calls. There are two functions provided for this purpose, one for raising a value to an *integer* power, and the other for raising a value to a real power (the latter is provided as a library function in TOP-SPEED Modula-2). The translated expressions preserve the right-to-left associativity of exponentiation in Fortran.

Another minor difference concerns the division operator. In Fortran, the real division operator and the integer division operator are syntactically the same while in Modula-2, '/' and 'DIV' are the real and integer division operators, respectively. The translator statically determines the type of the operands of the '/' operator in Fortran, and converts it to the 'DIV' operator only if both operands are of type integer.

5.3.1 Constant expressions

Both Fortran and Modula-2 support constant expressions. In Fortran, constant expressions can appear in constant definitions, array declarators, subscript expressions in EQUIVALENCE statements and DATA statements, substring length specifications in CHARACTER type statements and IMPLICIT statements, and in implied DO-loops in DATA statements. This in itself poses little problem as Modula-2 allows a constant ex-

pression wherever a constant value is required.

However, some restrictions have to be placed on the use of constant expressions in order for Fortran programs to be translatable. These restrictions stem from the fact that only Modula-2 operators and standard functions may appear in Modula-2 constant expressions, and therefore Fortran operations which are performed by translator defined procedures (such as exponentiation) cannot be performed in constant expressions. Consequently, Fortran constant expressions containing the exponentiation operator or complex arithmetic operations, cannot be translated.

On the other hand, Modula-2's type transfer functions are pre-defined and thus logical constant expressions and mixed mode arithmetic expressions can in general be translated (we will consider further restriction on constant expressions when EQUIVALENCE statements are discussed). As the evaluation of character string expressions is performed by library functions (see below), character string constant expressions cannot be translated.

5.3.2 Character string expressions

Fortran provides a string concatenatio operator. Moreover, Fortran allows substrings of string variables to be specified as both the target of assignments and as values. Modula-2 on the other hand does not provide any built-in string operators. Consequently, all string operations have to be performed by procedures. Most Modula-2 systems provide a library module which contains procedures for manipulating strings. Wirth does not specify what procedures should be included in a string handling module, and consequently they usually differ from system to system. Typically they contain procedures for assignment of string values to string variables[6], determining the length of a string[7], concatenating strings (normally returning the result in one of its arguments), comparing strings, inserting a string into a string, and extracting a substring from a string.

[6]Modula-2 is rather strict in its *assignment compatibility* requirement for strings. While a string literal can be assigned to any string variable which is *at least* as long as the literal, the value of a string variable can only be assigned to a string variable with the same length.

[7]Modula-2 uses a special *null* character to terminate strings which are shorter than the variable used to store them.

TopSpeed Modula-2 provides a string handling module called **str**. While the procedures in **str** are sufficiently powerful to perform virtually all Fortran string operations, only string assignment and string concatenation are handled by the translator to show how string operations in general can be translated.

The TopSpeed procedures for string assignment and string concatenation are described below:

```
  PROCEDURE Copy(VAR R: ARRAY OF CHAR; S: ARRAY OF CHAR);
(* copies string S to string R. If S is too long to fit,
   then the copy of S is truncated. If S is shorter than
   R then the string left in R is terminated by the null
   character *)

  PROCEDURE Concat(VAR R: ARRAY OF CHAR; S1,S2: ARRAY OF CHAR);
(* concatenates S1 and S2 and returns the result in R.
   The second string is truncated if the concatenated
   string becomes longer than the length of R *)
```

Now consider the following Fortran code[8]:

```
CHARACTER FIRST*10,SECOND*5,THIRD*5
...
FIRST = SECOND // THIRD
```

The translation of the assignment statement requires two procedure invocations and an additional variable to store the results of the concatenation operation before performing the assignment. Thus the above is translated into the following:

```
FIRST: ARRAY [0..9] OF CHAR;
SECOND: ARRAY [0..4] OF CHAR;
THIRD: ARRAY [0..4] OF CHAR;
Result: ARRAY [0..MaxLengthOfString-1] OF CHAR;
...
Concat(Result,SECOND,THIRD);
Copy(FIRST,Result);
```

where the constant **MaxLengthOfString** is defined as the length of the longest string variable in the Fortran program. Note that the use of zero indexing of the arrays **FIRST**,

[8] // is the concatenation operator in Fortran.

SECOND, and THIRD is not especially critical in this example. However, if the translator is to be expanded to handle other string operations, then the use of this convention is more important; all TopSpeed's string handling procedures assume that strings are zero indexed, including those which perform substring extraction and location of substrings within strings.

5.3.3 Intrinsic functions

Fortran provides a number of built-in or *intrinsic* functions which can (usually) be invoked within expressions ([ANS 78], pg. 15-2). While a few of them have counterparts in the Modula-2 *standard functions* (such as ABS and a few type conversion functions), most do not. Modula-2 in fact provides very little in the way of standard functions, although most Modula-2 systems provide libraries of useful functions. Typically, for instance, a library of mathematical functions is provided which contains trigonometric functions, a square root function, etc. These functions must be imported by modules which invoke them.

The main obstacle in the handling of intrinsic functions is that most of them are generic, taking arguments of various types. Library functions in Modula-2, on the other hand, are not. While it is possible in some cases to perform some type conversion of the arguments of intrinsic functions so that they conform to the types of the formal parameters of library functions, the translator instead only creates a *framework* for the user to define functions corresponding to intrinsic functions. That is, when a program unit contains an invocation of an intrinsic function, the module containing its translation imports a similarly named function from a module called Intrinsics: the definition of the function is left to the user.

This strategy will be correct only if the specific names of intrinsic functions are used in the Fortran program since generic function names usually represent several functions. For instance, the Fortran generic function name MOD actually represents three specific functions: MOD (which takes integer arguments), AMOD (which takes real arguments), and DMOD (which takes double precision arguments). A more complete version of the translator could

provide function definitions in `Intrinsics` for each specific intrinsic function[9] and convert generic function calls into invocations of these functions (the particular function invoked would be determined by the types of the arguments supplied to the generic function).

5.4 Specification statements

5.4.1 DIMENSION statements

The DIMENSION statement ([ANS 78], pg. 8-1) is used to specify the names of arrays along with the number and size of their dimensions. Arrays in Fortran are translated into arrays in Modula-2, and the translator generally preserves the subscript ranges of the Fortran arrays. The sole exception to this is in the treatment of arrays of character strings for which an additional dimension is provided.

Subscripts in Fortran must be of type INTEGER, and similarly subscript ranges and values in their Modula-2 translation are of type INTEGER. Modula-2's characteristic strictness of type checking manifests itself in the use of subscript range specification and array element reference, and the type of subscript expressions must agree exactly with the type used in the subscript range specification (ie., the *index type* of a given array). For instance, the following declarations:

```
IND: INTEGER;
ARR: ARRAY [1..10] OF REAL;
```

are incorrect for the element reference `ARR[IND]`; the problem here is that when a Modula-2 compiler sees the declaration of `ARR` it treats the index type to be a subrange of type `CARDINAL`[10]. In order to specify that the index type is a subrange of INTEGER values, the array would have to be declared as follows:

```
ARR: ARRAY INTEGER[1..10] OF REAL;
```

[9]The Fortran intrinsic functions `MIN` and `MAX` which take a variable number of arguments could be converted into a "nested sequence" of two-argument function invocations to permit their translation into mboxModula-2

[10]Naturally this problem would not have occurred had the lower bound in the dimension specification been a negative number.

The translator, in fact, specifies index types in this manner. Since INTEGER type in Fortran is mapped onto LONGINT type in (TopSpeed) Modula-2, it might seem more convenient to use index types which are subranges of type LONGINT. But alas, TopSpeed does not permit LONGINT values to be used to subscript arrays. Subsequently, in the translation the type transfer function INTEGER is used to convert subscript expressions to type INTEGER; in the above example, a Fortran reference to `ARR(IND)` is converted to the Modula-2 reference `ARR[INTEGER(IND)]`.

In the examples presented in the following sections, this conversion of subscript ranges and expressions is not shown to keep the examples uncluttered.

Some additional discussion (and restrictions) on the use of multi-dimensional arrays in the translation of EQUIVALENCE statements, COMMON statements, and dummy arguments is given in later sections.

5.4.2 IMPLICIT statements and type statements

The translator handles both IMPLICIT statements ([ANS 78], pg. 8-7) and type statements ([ANS 78], pg. 8-5). Any specified implicit typing represented by the former is stored in an attribute which is later used in determining the types of entities which are not explicitly typed (this attribute also includes a description of Fortran default implicit typing convention).

When the translator encounters a type statement, it associates with the entities named in the statement the type specified in the type statement. This information is stored in an attribute which represents the *environment* of the program unit. In general, type statements only associate names with types. Such names may be used to name variables, constants, arrays, functions, or dummy arguments, but further description of the entity associated with a name normally appears elsewhere in the program[11]. The exception is

[11] In fact, the same name can be used for *several* entities in a given program unit. For instance, dummy argument names in a statement function need not be distinct from variable names in a program unit. The appearance of a given name in a type statement simply asserts that any entity referred to by that name in the same program unit shall be of the specified type.

when names of arrays appear in type statements, since the type statement *may* specify the dimensions of the array. In this case, the translator includes both the type and dimensionality of the array in the environment attribute (a similar approach is taken when an array dimensionality is specified in a COMMON statement).

5.4.3 COMMON statements

The COMMON statement in Fortran provides a means of associating entities in different program units by defining common blocks of storage. This allows different program units to define and reference the same data without using argument passing mechanisms, and to share storage units ([ANS 78], pg 8-7).

Common blocks of storage are translated into Modula-2 record variables which are shared amongst modules. Such a record variables' fields correspond to entities in the common block. A reference to an entity in a common block thus becomes a qualified reference to the corresponding record in the translated program.

In the simplest case, all program units sharing a common block define it in precisely the same way. For instance, suppose all program units of an executable Fortran program contain the following statements:

```
REAL Q
INTEGER A(10),B(5)
COMMON /CBLK/ A,Q,B
```

The translation of this is straight-forward. A record variable `CBLK` is declared in the Modula-2 program as follows:

```
CBLK : RECORD
         A: ARRAY [1..10] OF LONGINT ;
         Q: REAL ;
         B: ARRAY [1..5]  OF LONGINT
       END;
```

In each program unit, applied occurrences of `A` would be translated into references to `CBLK.A` , applied occurrences of `Q` would be translated into references to `CBLK.Q`, and so

on. In order to make the generated record accessible to all program units that share it, the generated record variables corresponding to common blocks are declared in a separately compiled module called COMMON. Any modules containing subprograms which define and reference common blocks must IMPORT the appropriate record variables from COMMON.

A drawback to this method is that the definition of common block entities lies outside of the program units, although the interested human reader is referred to the definition module of COMMON by the import list. One remedy would be the inclusion of a comment in modules importing structures from COMMON which shows declarations of such structures as they appear in COMMON.

When the names of entities in common blocks differ from unit to unit, the above approach is insufficient, since different names may be used to refer to the same storage unit(s) in a block. For example, suppose common block CBLK were defined as follows[12]:

COMMON /CBLK/ A(10),B(5) in subprogram S1, and as

COMMON /CBLK/ X(5),Y(10) in subprogram S2.

The strategy employed in the translator for such cases is to generate variant record fields in the record variable, each variant being associated with a particular subprograms *image* of the common block. For example, the above common block could be translated into the following record variable:

```
CBLK : RECORD
  CASE tag: CARDINAL OF
    1: A: ARRAY [1..10] OF REAL;
       B: ARRAY [1..5] OF REAL |
    2: X: ARRAY [1..5] OF REAL ;
       Y: ARRAY [1..10] OF REAL
  END
END;
```

Subprogram S1 would refer to the element A[6] as CBLK.A[6] while subprogram S2 would

[12]Default implicit typing is assumed to apply to the typing of non-explicitly typed entities, such as A, B, X, and Y in this example.

refer to the same location as `CBLK.Y[1]`[13].

Note that the correctness of this strategy ([Fre 81]) depends on the Modula-2 compiler in the target environment overlaying variants in memory (as does the TopSpeed compiler). The key to the semantic correctness of the translation lies in the sharing of the *first unit of storage* of the variants.

In Modula-2, field names must be distinct from all other field names in the record, including those fields outside the variant part, fields in other variants, and the tag field itself. So when the same name is used in different program units to refer to (possibly) different storage units within a common block, the field names must be made distinct. The translator implements this by *tagging* the names of entities in common blocks with a number which is uniquely associated with the program unit in which it is defined.

For example, if common block CBLK were defined as

`COMMON /CBLK/ A(10),B(5),C(5)` in subprogram Sa, and as

`COMMON /CBLK/ C(8),Y(10),A(2)` in subprogram Sb,

then the corresponding structure in the translation (assuming that 1 was the value uniquely associated with subprogram Sa and 2 was the the value associated with subprogram Sb) would be:

```
CBLK : RECORD
  CASE tag: CARDINAL OF
    1: A_1: ARRAY [1..10] OF REAL;
       B_1: ARRAY [1..5]  OF REAL;
       C_1: ARRAY [1..5]  OF REAL |
    2: C_2: ARRAY [1..8]  OF REAL;
       Y_2: ARRAY [1..10] OF REAL;
       A_2: ARRAY [1..2]  OF REAL
  END
END;
```

Note that the underscore (_) is not in the standard Fortran-77 character set, and thus no name conflicts will arise as a result of this tagging strategy. Moreover, Modula-2 is *case*

[13]The translator actually generates variant records with *tag fields* which are of TopSpeed Modula-2 type `SHORTCARD` (a one byte version of `CARDINAL`).

sensitive, and since the translator keeps Fortran names upper-case (and only modifies the Fortran names - when necessary - by appending characters to them), generated identifiers such as `tag` are also guaranteed to be distinct from names in the Fortran program.

Another area of concern is the compilers' treatment of the *tag fields* in variant records. Normally, a program should not reference the fields within a variant part until a value has been assigned to the tag field, following which the variant corresponding to that value becomes *active*. In the above example, if the value of `tag` were 1, then the first variant would be active and references to `CBLK.A` and `CBLK.B` would be considered valid, while references to `CBLK.X` or `CBLK.Y` would not. When a program attempts to reference a field in a variant which is not active, a Modula-2 system should *theoretically* check the tag field before making the reference. However, for efficiency, this check is often not performed[14]. This permits the current translator to ignore the existence of the tag field when translating references to common block members. This reliance on the peculiarities of a Modula-2 compiler makes the translation strategy *non-standard*, but even without the absence of tag-checking, the strategy could be employed provided some additional assignments to such tag fields were made.

Within an executable Fortran program, all common blocks which have the same name must have the same size, with the exception of blank common. Thus in the case of the record corresponding to blank common block, variants may be of different sizes. This can be handled in Modula-2 if the compiler in the target environment allocates enough space for the largest variant ([Kin2 88]).

5.4.4 EQUIVALENCE statements

The EQUIVALENCE statement in Fortran is used to specify the sharing of storage locations by two or more entities in a program unit ([ANS 78], pg. 8-1). The difference between common block storage locations and locations shared by *equivalenced entities* (ie., entities which are associated with each other *via* EQUIVALENCE statements) is that the

[14] As in TopSpeed Modula-2, since it must be performed at run time.

former are shared by entities from different program units, while the latter are shared by entities within the same program unit (of course, entities in a common block can be equivalenced with other entities within a given program unit; this case will be discussed shortly).

The strategy for dealing with equivalenced entities is much the same as that with which common blocks were handled. Consider the (simplest) case where no equivalenced entities are present in any common blocks. In the translation of such program units, a variant record would be generated, each variant corresponding to a particular equivalenced entity. For instance, if a program unit contains

```
INTEGER NUMBERS(100)
REAL RANGE(100)
EQUIVALENCE (RANGE,NUMBERS)
```

then in the translation, the following record variable is generated:

```
eqclass1 : RECORD
             CASE tag: CARDINAL OF
              1: NUMBERS: ARRAY [1..100] OF LONGINT |
              2: RANGE: ARRAY [1..100] OF REAL
             END
           END;
```

All references to RANGE in the Fortran program are translated into Modula-2 references to `eqclass1.RANGE`. Here as well, the correctness of the translation hinges on the Modula-2 compilers' overlaying of variants.

In the example above, NUMBERS and RANGE have the exact same storage sequence, and are thus said to be *totally associated*. Entities are said to be *partially associated* if they share *some* but not all of their storage sequence. An example is the following:

```
INTEGER NUMBERS(50)
REAL RANGE(100)
DOUBLE PRECISION Q
EQUIVALENCE (NUMBERS,Q,RANGE)
```

These statements specify that NUMBERS, Q, and RANGE share the same *first* storage unit. Their translation into Modula-2 is the following:

```
eqclass2 : RECORD
             CASE tag:CARDINAL OF
               1: NUMBERS: ARRAY [1..50] OF LONGINT |
               2: Q: LONGREAL |
               3: RANGE: ARRAY [1..100] OF REAL
             END
           END;
```

It is also possible that partially associated entities named in an equivalence statement do not share the same first storage unit, as in the following example:

```
INTEGER NUMBERS(50), RANGE(50)
EQUIVALENCE (NUMBERS,RANGE(10))
```

In this example, the first storage unit in the sequence for NUMBERS is the same as (associated with) the tenth unit in the storage sequence for RANGE. In such a case, the translator includes *padding* of some variant fields. For example, the above would be translated into the following:

```
eqclass3 : RECORD
            CASE tag:CARDINAL OF
             1: displacement_1: ARRAY [1..9] OF LONGINT;
                NUMBERS:        ARRAY [1..50] OF LONGINT |
             2: RANGE:          ARRAY [1..50] OF REAL
            END
           END;
```

Another example follows; here association of entities of different types is specified (Fortran permits such associations, provided character string entities are not associated with entities of any other type).

```
REAL X(100)
LOGICAL BITS(8)
DOUBLE PRECISION BIGNUM(25)
EQUIVALENCE (BITS(3),X), (X(5),BIGNUM(10))
```

Another feature of this example is that not all associated entities are present in the same *equivalence-list*. The corresponding record declaration is the following (recall that Logical is the translator defined type which is equivalent to type LONGWORD):

```
eqclass4: RECORD
      CASE tag:CARDINAL OF
        1: BIGNUM:         ARRAY [1..25]  OF LONGREAL |
        2: displacement_2: ARRAY [1..12]  OF LONGWORD;
           BITS:           ARRAY [1..8]   OF Logical  |
        3: displacement_3: ARRAY [1..14]  OF LONGWORD;
           X:              ARRAY [1..100] OF REAL
      END
   END;
```

A difficulty arises when this strategy is implemented. Namely, when constant expressions are used in the array declarators for array(s) appearing in EQUIVALENCE statements or when constant expressions are used to subscript an entity in an EQUIVALENCE statement, these expressions must be evaluated by the translator in order to compute the various offsets. Subsequently, if such constant expressions are translated into constant expressions in the Modula-2 program, some of the correctness of the translation is compromised, since modification of any of the constants in the constant expressions in the Modula-2 program will *not* in fact result in any change to the corresponding record structures, even though the same change in translated Fortran program could result in different storage associations.

Presently, the translator *does* evaluate constant expressions which appear in array declarators and array element references in EQUIVALENCE statements, and code for such constant expressions is not generated; only the values computed (namely, integer constants) appear in the translated programs. Since the translator evaluates integer constant expressions, additional constraints have to be imposed on their form in Fortran programs because ALADIN does not provide any numerical type other than integer type (it does provide facilities for performing arithmetic operations on integers). Thus, while Fortran allows arithmetic constant expressions wherever integer constant expressions may appear, the translator constrains integer constant expressions to contain *only* integer operands.

Another difficulty concerns the treatment of multi-dimensional arrays. In Fortran, the same sequence of locations may be interpreted as a one-dimensional array by one entity and as a multi-dimensional array by another. Because of the different storage allocation

schemes for multi-dimensional arrays used in Fortran and Modula-2 (column-wise vs. row-wise), it would not be sufficient (or correct) to simply have one variant in the form of a one-dimensional array, and the other in the form of a multi-dimensional array[15]. For instance, suppose a Fortran program unit contained the following specification statements:

```
INTEGER A(25),B(5,5)
EQUIVALENCE (A,B)
```

In the program unit, both `A(5)` and `B(5,1)` are references to the fifth location in the shared storage sequence. If this were translated into the following:

```
eqclass : RECORD
              CASE tag:INTEGER OF
                 1: A: ARRAY [1..25] OF INTEGER |
                 2: B: ARRAY [1..5,1..5] OF INTEGER
              END
           END;
```

then the fifth memory location in `eqclass` would be shared by `A[5]` and `B[1,5]`, which clearly is not equivalent semantically to the Fortran version.

Consequently, all multidimensional arrays in the Fortran source are being converted into one dimensional arrays. At present, the translator simply assumes that this conversion has been done prior to translation (as a preprocessing step). Further discussion of arrays is given in the concluding chapter.

Use of EQUIVALENCE Statements with COMMON Statements

When an entity is associated with an entity in a common block, references to the associated entity are in effect references to storage unit(s) within the block. The translator handles such associations by including additional variant fields in the blocks' record for each of the entities associated with entities in the block. Such variants may include some padding which reflects the offsets of associated entities from the start of the block[16].

---

[15] This problem in fact arises whenever associated arrays are dimensioned differently.

[16] Fortran forbids associations which extend the common block storage sequence by adding storage units which precede the first storage unit of the first entity specified in a common statement for the common block([ANS 78], pg. 8-5).

Consider the following example:

```
INTEGER A(10),B(10),C(10)
DOUBLE PRECISION DP(10)
COMMON /CBLK/ A,B
EQUIVALENCE (C(5),B), (DP(2),C(2))
```

Assuming that CBLK is defined identically in all program units, the record declaration corresponding to the common block is the following:

```
CBLK : RECORD
        CASE tag:CARDINAL OF
        1: A:  ARRAY [1..10] OF LONGINT;
           B:  ARRAY [1..10] OF LONGINT   |
        2: displacement_2: ARRAY [1..6] OF LONGWORD;
           C:  ARRAY [1..10] OF LONGINT
        3: displacement_3: ARRAY [1..5] OF LONGWORD;
           DP: ARRAY [1..10] OF LONGREAL  |
        END
       END;
```

A separate record declaration is of course not generated for the equivalence class.

The number of storage units by which an associated entity is offset from the start of the block with which it is associated is given by:

$$offset(e_i) = displacement(e_i) + (offset(e_k) - displacement(e_k))$$

where e_i is the associated entity, e_k is the entity associated with e_i which appears in a COMMON statement, $offset(e_k)$ is the offset of e_k from the start of the block, and $displacement(e_j)$ is the offset of entity e_j from the start of the storage sequence with which e_j has been associated through EQUIVALENCE statements *only*.

5.4.5 SAVE statements

The SAVE statement in Fortran is used to specify that the definition status of entities (variables, array variables, and named common blocks) is to be *saved* (retained) after the execution of a RETURN or END statement in a subprogram ([ANS 78], pg. 8-10). A SAVE statement has no effect in the main program, and thus is ignored in its

translation. In the translation of subprograms, the translator handles saved entities by declaring them *outside* of the corresponding procedure or function definition but *inside* the implementation module containing the translation of the subprogram. That is, saved entities are made *global* to the subprogram that references them but *local* to the module containing the subprograms' definition. Thus, while such variables can only be referenced within the implementation module, once they become defined, their value is maintained across procedure calls.

As common blocks are declared and exported by the module `COMMON`, their definition status is never altered by returns from subprograms in the Modula-2 program. Thus no special action is taken when common block names appear in SAVE statements. If an entity associated with a common block is named in a SAVE statement, again no special action is taken.

If an entity which is associated with other entities through EQUIVALENCE statements but *not* associated with a COMMON block appears in a SAVE statement, all its associated entities are also saved by declaring the corresponding record structure in the implementation module containing the subprogram's definition.

5.4.6 EXTERNAL statements

In Fortran, the EXTERNAL statement when it appears in a given program unit is used to declare symbolic names in the unit as representing either an external procedure, a dummy procedure (a dummy argument which represents a procedure), or a block data subprogram. A dummy procedure name which appears in an EXTERNAL statement may be used as an actual argument ([ANS 78], pg. 8-9).

In Modula-2, procedures can be supplied as arguments in calls to other procedures. Thus, modules which contain translations of subprograms that declare subprograms as external procedures need simply import their identifiers from the modules containing their definition thereby allowing the *external* procedure to be passed as argument. Of course, modules which contain translations of subprograms which invoke external procedures have

to import them whether or not the procedure names appear in an EXTERNAL statement.

For example, consider the following Fortran program unit:

```
PROGRAM P
EXTERNAL SUB1
...
CALL SUB0(SUB1)
...
END
```

It is translated as the following[17]:

```
MODULE P;
(* all external procedures must be imported *)
FROM SUB1_mod IMPORT SUB1;
FROM SUB0_mod IMPORT SUB0;
...
SUB0(SUB1)
...
END P.
```

A more difficult problem concerns the translation of subprograms with dummy procedures. Modula-2 enforces strict type checking rules, and actual arguments must correspond exactly in type with formal parameters. Consequently, in the corresponding Modula-2 procedure, type identifiers must be available which represent the types of the procedures which are passed as argument.

This situation is less troublesome when the dummy procedure is called or invoked from within the subprogram in which it appears as a dummy argument; since Fortran requires that the types of the actual arguments in the invocation of the dummy procedure must correspond exactly with the types of the dummy arguments of the subprogram represented by the dummy procedure, generation of the appropriate procedure type can be done by inspection of the actual arguments supplied to the dummy procedure. Moreover, it can be determined by the context of the invocation whether the subprogram is a subroutine

[17]The names of modules used to define external subprograms are based on the name of the procedure or function principally defined by it: they are formed by appending the string '_mod' to the procedure name.

or a function. In the latter case, the type of the function can be determined by either the occurrence of the dummy procedure name in a type statement, or by implicit typing of the function name[18].

However, if a dummy procedure appears only in an EXTERNAL statement and as an actual argument, it is impossible to determine the number or type of the dummy arguments of the dummy procedure, or whether for that matter the dummy procedure is a function or a subroutine ([ANS 78], pg. 15-9). To reduce the complexity of the translator and allow the treatment of dummy procedures to be handled locally (ie., by inspection of the subprogram unit only) the translator forbids this situation, requiring that subprograms with dummy procedures contain at least one invocation of each dummy procedure declared.

For example, suppose the definition of SUB0 of the previous example was the following:

```
SUBROUTINE SUB1(SUB2)
EXTERNAL SUB2,SUB3
...
CALL SUB2(X,Y)
CALL SUB3(SUB2)
...
END
```

Here, SUB2 is a dummy procedure which is both invoked and passed as argument to SUB3. It would be translated as follows:

```
DEFINITION MODULE SUB1_mod;
TYPE SUB2_type = PROCEDURE (VAR REAL ,REAL);
...
END SUB1.

IMPLEMENTATION MODULE SUB1_mod;
FROM SUB3_mod IMPORT SUB3;
...
```

[18] If an actual argument is an intrinsic function, it never has any automatic typing property ([ANS 78], pg. 15-9). This does not create any problem for the translator, however, since intrinsic functions can only be passed as argument by program units which name them in an INTRINSIC statement, and the INTRINSIC statement is not translated.

```
PROCEDURE SUB1 (SUB2: SUB2_type);
...
SUB2(X,Y);
SUB3(SUB2);
...
END SUB1;
END SUB1_mod.
```

Notice that the formal parameter SUB2 actually functions as a procedure variable within SUB1 and is passed by value. In order to satisfy the requirement that actual and formal parameters match exactly in Modula-2, the type identifier used to define the type of the formal parameter in the definition of SUB3 would have to have the *same defining occurrence* as that used in the formal parameter list of SUB3. For this reason, in the code generated for SUB1, the definition of SUB2_type does not actually appear in the definition module of SUB1 as shown above. Instead, it would appear (along with the definition of all translator generated type identifiers) in a module called Types and thus a type equivalent to SUB2_type would actually be imported from Types by SUB1_mod.

The appearance of a block data subprogram name in an EXTERNAL statement is ignored by the translator since it primarily indicates that the block data subprogram name is the same as that of an intrinsic function ([Mei 82]), thereby making the intrinsic function with the same name unavailable within the subprogram containing the EXTERNAL statement. Thus no special action is taken when a block data subprogram name appears in an EXTERNAL statement, since in the translation block data subprogram names are not actually used (the initialization of common blocks that they specify is performed in the module COMMON).

When an intrinsic function name appears in an EXTERNAL statement in a program unit, that name becomes the name of an external procedure and the equivalently named intrinsic function is unavailable in that program unit. Thus, Modula-2 functions corresponding to intrinsic functions are only imported from Intrinsics by modules containing the translation of program units which invoke - *but do not declare as* EXTERNAL - intrinsic functions.

5.4.7 DATA statements

The DATA statement in Fortran is used to provide initial values for variables, arrays, and array elements at the start of an executable program ([ANS 78], pg. 9-1). In the translation, DATA statements are converted into assignment statements. Entities appearing in DATA statements in the main program are initialized by assignment statements which appear at the start of the main module body of the Modula-2 program (before any statements corresponding to Fortran executable statements).

Entities initially defined in DATA statements within subprogram units are *SAVE*d in Fortran. Consequently such entities are declared in the same manner as explicitly *SAVE*d entities in the module containing the translation of the subprogram. Assignment statements in the module body initialize these entities. Since module bodies of separately compiled modules are executed prior to execution of the main program body, this guarantees that the initialization is performed before those actions corresponding to Fortran executable statements.

Two restrictions have been placed on the use of DATA statements, one is included only to ease the job of the translator (and can easily be removed) while the other is more serious and arises from the use of named constants in DATA statements. The more serious restriction is discussed first.

The DATA statement in Fortran has the following general form[19]([ANS 78], pg. 9-1]):

```
DATA <nlist> /<clist>/ { [,] <nlist> /<clist>/ }
```

where `<nlist>` is a list of variable names, array names, array element names, substring names, and implied-DO lists[20], and `clist` is a list each element of which has the form `c` or `r*c` where `c` is a constant and `r` (referred to below as the *repeat count*) is a positive integer constant or named positive integer constant.

[19]The notation used throughout the paper to describe the syntax of Fortran constructs is in the style of EBNF: terminal symbols are written without modification (eg., symbols DATA and the slash (/) in the description of the syntax of the DATA statement), nonterminal symbols are enclosed within angle brackets (<,>), syntactic entities within curly brackets ({,}) indicate zero or more occurrences of those entities, and syntactic entities within square brackets ([,]) indicate zero or one occurrences.

[20]The translation of implied-DO lists in DATA statements has not been implemented.

The translator imposes the following restriction on the form of <nlist>: the list of names must either contain a single array name (all of whose elements are to be defined by values in the constants list) or one or more variables or array elements (in which case the constants list contains a constant for every element reference or variable in the name list). Moreover (and this is the less serious restriction), at most a single repeat count may appear in the constant list, and it *must* be equal to the number of locations to be initialized implied by the name list.

For example, the following cases can be handled by the translator[21]:

```
PARAMETER (N=10)
INTEGER A(N),B(100),C(N-5)
LOGICAL FLAG, FOUND
DATA A/N*0/, FLAG,FOUND,B(3)/.TRUE.,.FALSE.,25/
DATA C/1,2,3,4,5/, X,Y,Z/3*0/
```

The assignment statements which perform the required initialization are generated as follows:

```
FOR repeatcount := 1 TO N DO
  A[repeatcount] := 0
END;
FLAG := Logical(TRUE);
FOUND := Logical(FALSE);
B[3] := 25;
C[1] := 1;
C[2] := 2;
C[3] := 3;
C[4] := 4;
C[5] := 5;
X := REAL(0);
Y := REAL(0);
Z := REAL(0);
```

When a single repeat count appears in the list of constants specifying initial values for elements of an array, a straightforward implementation using a FOR loop can be realized. Also, in the initialization of X, Y, and Z the type of the value to which the variables are

[21]The PARAMETER statement in Fortran defines named constants: they are converted by the translator into Modula-2 constant definitions.

initialized differs from the type of the variables, and thus type transfer functions are used to convert the integer constant 0 to the type REAL.

A situation that is avoided by imposing the restriction on the name list is demonstrated in the following example:

```
PARAMETER (N=5,M=2)
INTEGER A(N), B(M)
DATA A,B/2,2,1,3,4,4,7/
```

In this case, a loop cannot be used for the translation of the initialization of A, and something like the following assignments would be called for in the translation:

```
A[1] := 2; A[2]:= 2; A[3] := 1; A[4] := 3; A[5] := 4;
B[1] := 4; B[2] := 7;
```

However, if the values of N and M were changed to become 2 and 5 respectively, then the above code would no longer be equivalent. By constricting the name list to have only a single array name or none at all, this situation is avoided.

The restriction of allowing only one repeat count in the constants list is introduced to simplify the translation, and could easily be removed. For instance, the following situation could be handled with only slight modification to the translator:

```
PARAMETER (N=5,M=4)
INTEGER A(10)
DATA A/N*0,80,M*5/
```

The initialization of A could be accomplished by the following assignments:

```
FOR repeatcount := 1 TO N DO
    A[repeatcount] := 0
END;
A[N+1] := 80;
FOR repeatcount := N+2 TO 10  DO
    A[repeatcount] := 5
END;
```

DATA statements in block data subprograms

In Fortran, block data subprograms are used to predefine values in named common blocks. Such subprograms contain only declarations and are global to the executable program ([ANS 78], pg. 16-1). In the translation, no separate module is generated for block data subprograms: the initialization of common block entities specified by DATA statements within block data subprograms is performed by assignment statements in the module body of the module `COMMON`.

5.4.8 Statement functions

The *statement function* in Fortran is a function that is defined by a single (non-executable) statement within a program unit ([ANS 78], pg. 15-4). Statement function *definitions* have the following general form:

```
<name> ( <list of dummy arguments> ) =  <expr>
```

Statement functions are analogous to external function subprograms whose executable statement part consists of a single assignment statement (in which the function name appears on the left hand side of the assignment operator).

The scope of a statement function is restricted to the part of the program unit in which it is defined *following* the line on which it is defined (statement function definitions must appear prior to the executable statement part of a program unit, but their invocation may appear in the expressions of any *subsequent* statement function definitions). The translator converts statement functions into Modula-2 functions. The function definitions for statement functions in the main program unit are defined within the main program module, while those defined in external procedures are nested within the definition of the corresponding procedure or function. As Modula-2 does *not* require that procedures be defined before they are called, no special importance is given to the order in which the definitions of translated statement functions appear.

The dummy argument list in a statement function definition is a (possibly empty) list of variable names which serves only to indicate the type, order, and number of arguments for

the statement function. Their scope is limited to the statement function itself. Primaries of the expression in a statement function definition may be one of the following: a (named or unnamed) constant, a variable reference, an array element reference, an intrinsic function reference, a reference to a previously defined statement function, an external function reference, a dummy procedure reference, or a parenthesized expression meeting all the above requirements.

The formal parameter list corresponding to the dummy argument list in the function procedure constructed by the translator consist of *value* parameters. While the Fortran-77 standard does not directly stipulate that the values of actual arguments (which are variables or array elements) to a statement function cannot be modified, it does stipulate that an external function reference must not cause a dummy argument of a statement function to become undefined or redefined. Since external function references are the only possible means of modifying the values of dummy arguments (and hence, actual arguments) of a statement function, the formal parameter list generated for the dummy argument list in the translation of statement functions consists solely of *value* parameters.

For example, consider the following subprogram which contains two statement function definitions:

```
SUBROUTINE SAMPLE( ... )
...
SUMSQ (X, Y, Z) = X*X + Y*Y + Z*Z
MYFUN (A,B,C) =  2 * SUMSQ(A,B,C) / 3
...
```

The translation of **SAMPLE** is as follows:

```
PROCEDURE SAMPLE( ... );
...
 PROCEDURE SUMSQ (X,Y,Z: REAL) : REAL;
   BEGIN
    RETURN X*X + Y*Y + Z*Z
   END SUMSQ;
 PROCEDURE MYFUN (A,B,C: REAL) : LONGINT;
   BEGIN
     RETURN LONGINT( REAL(2) * SUMSQ(A,B,C) / REAL(3))
```

```
    END MYFUN;
...
```

The RETURN statement in Modula-2, when executed, indicates the value to be returned by a function, as well as terminating function execution[22]. Note also that the type of the expression may have to be converted (in accordance with Fortran assignment rules) to the type of the function.

The expressions in statement function definitions may also contain references to objects which do not appear in the dummy argument list, and this is handled rather easily by the translator. The following example is of this kind:

```
REAL A,B, SFUN
COMMON /BLK/ A(100), B(100)
SFUN (A) = A + B(12)
```

In this example, the array B is global to SFUN, while within the function expression the name A refers to the dummy argument. The translation of SFUN would be the following (ignoring the tagging of names of entities in common blocks):

```
PROCEDURE SFUN (A: REAL) : REAL;
  BEGIN
    RETURN A + BLK.B[12]
  END SFUN;
```

5.5 Executable statements

5.5.1 DO statements

The Fortran DO statement is used for specifying definite iteration, and as such its counterpart in Modula-2 is the FOR statement (which is similar though not identical to the FOR loop in PASCAL). While the heart of the translation of a given DO loop consists of a FOR loop, a number of syntactic and semantic differences between the two necessitates the addition of extra variables and statements.

[22]The RETURN statement may also be used in procedures where its' execution causes procedure termination (the expression is absent from the syntax of the statement when used in a procedure).

The Fortran DO loop has the following form:

```
      DO <label> [,] <var> = <expr1>, <expr2> [, <expr3>]
         <stmt-sequence>
<label>  <stmt>
```

where <label> is the statement label of the *terminal statement* of the DO loop (the occurrences of the nonterminal <label> must be identical) , <var> is the integer, real, or double precision type control variable of the loop, and <expr1>, <expr2>, and <expr3> are integer, real, or double precision expressions used to specify the *starting value*, *limit*, and *increment*, respectively.

The Modula-2 FOR loop has the following general form:

```
FOR <ident> := <expr1> TO <expr2> [ BY <const_expr> ] DO
   <stmt-seq>
END
```

where <ident> is an ordinal typed variable (ie., it cannot be of real or longreal type), <expr1> and <expr2> are expressions which must be *compatible* with the type of <ident>, <const_expr> is a constant expression of a type compatible with that of <ident>, and <stmt-seq> is a sequence of statements.

The restrictions on the FOR loop that the type of the control variable be an ordinal type and that any specified increment must be a constant expression makes it impossible to translate the DO structure using a FOR loop alone. While it would be possible to implement the DO loop with a Modula-2 WHILE loop, the FOR loop was chosen because in the formal semantics of the DO loop, an iteration counter variable independent of the DO loop control variable is used to control the iteration ([ANS 78], pg. 11-7). The control variable of the FOR loop in the Modula-2 translation of a DO loop corresponds to this independent counter.

The strategy for translating DO loops is summarized in the following: (up to) five auxiliary variables are generated by the translator for the purposes of storing the results of evaluation of the expressions giving the start value (`StartValue`), limit (`Limit`), and increment (`Incr`), as well as a variable for storing the number of iterations of the loop

(IterationCount) and a control variable for the FOR loop (InterationCtrl). The type of StartValue, Limit, and Incr (if present) is the same as that of the control variable of the DO loop. IterationCount and IterationCtrl are of type LONGINT[23].

```
(* evaluate expr1, expr2, and (possibly) expr3 *)
(* T is the type of the DO loop ctrl variable *)
   StartValue := T(expr1);
   Limit := T(expr2);
   Incr := T(expr3);
(* initialize DO variable *)
   do_var := StartValue;
(* determine number of iterations *)
   IterationCount:= LONGINT((Limit-StartValue)/Incr + T(1));
(* use FOR to perform iteration *)
   FOR IterationCtrl := 1 TO IterationCount DO
      translation(<stmt-sequence>);
      do_var := do_var + Incr
   END;
```

For the most part the above strategy follows the semantics of the DO loop. The variable IterationCtrl corresponds to the internal Fortran variable used to control definite iteration, and is kept distinct from all other variables in the code for the loop. In Modula-2 the behavior of a FOR loop is considered unpredictable if either the start value,the limit value, or the control variable is modified by statement(s) in the loop. This constraint prompted the use of IterationCtrl as the control variable in the FOR loop, and the introduction of IterationCount to store the limit value. StartValue, Limit, and Incr had to be introduced to account for possible side-effects of <expr1>, <expr2>, and <expr3>, and to preserve the order of evaluation of the expressions[24].

It should be noted that the type transfer function LONGINT truncates real values and thus the expression for computing the number of iterations is equivalent the the Fortran

[23] Some Modula-2 systems disallow control variables of FOR loops to be of type LONGINT, but TopSpeed permits it!

[24] Clearly, much more efficient code can be generated for DO loops in which the start, limit, and/or increment expressions do not have side-effects; in such cases, variables StartValue, Limit, and/or Incr need not be introduced. Moreover, if constants or constant expressions are used, then IterationCount need not be used. The translator does not currently attempt such optimization, but could easily be extended to do so.

expression for the iteration count given by `MAX(INT((Limit - StartValue + Incr) / Incr),0)` ([ANS 78]).

It should also be noted that when the type of the DO variable is integer, the division operator used in the calculation of the iteration count is `DIV`. Moreover, when no increment is specified the iteration count is computed using `Limit - StartValue + 1` when start value and limit are integers and using `LONGINT(Limit - StartValue + 1.0)` otherwise.

When DO loops are nested, it is necessary to make some of the auxiliary identifiers distinct (at least `Incr`, `IterationCount`, and `IterationCtrl`). This is done by appending numerical suffixes onto these identifiers which reflect the depth of nesting of the particular loop they are being used to implement. For instance, the loop

```
      DO 100 I = 1,10
         DO 200 J = 1,I
            SUM = SUM + I
200      CONTINUE
100   CONTINUE
```

is translated into:

```
StartValue:= 1; Limit:= 10; I:= StartValue;
IterationCount_1 := LONGINT(Limit - StartValue + 1);
FOR IterationCtrl_1 := 1 TO IterationCount_1 DO
   StartValue:= 1; Limit:= I; J:= StartValue;
   IterationCount_2 := LONGINT(Limit - StartValue + 1);
   FOR IterationCtrl_2 := 1 TO IterationCount_2 DO
     SUM := SUM + I;
     J := J + 1;
   END
   I := I + 1;
END
```

The strategy of suffixing auxiliary identifiers by numerical suffixes alone is however insufficient in general, since loops at the same level of nesting would then share some of the same auxiliary variables. But as the type of the control variables for distinct DO loops at the same level of nesting may differ, further differentiation of `StartValue`, `Limit`, and `Incr`, is called for.

Consequently, the translator appends an additional suffix to these identifiers when they are used to store real or double precision values (the characters 'r' and 'd' respectively). The advantage of this strategy is that these variables can be reused in the code for loops at the same level of nesting with the same type of control variable.

A further example is provided to demonstrate this identifier generation strategy:

```
      DO 100 X = 1.0, 5.5, 2.0
         SUM = SUM + X
100   CONTINUE
      DO 200 I = 1,10
         SUM = SUM + I
200   CONTINUE
```

Its translation is the following:

```
Start_r:= 1.0; Limit_r:= 5.5; Incr_r_1:= 2.0;
X := Start_r;
IterationCount_1:= LONGINT((Limit_r - Start_r)/Incr_r_1 + 1.0);
FOR IterationCtrl_1 := 1 TO IterationCount_1 DO
  SUM := SUM + X;
  X := X + Incr_r_1;
END;

Start:= 1; Limit:= 10; I := Start;
IterationCount_1 := LONGINT(Limit - Start + 1);
FOR IterationCtrl_1 := 1 TO IterationCount_1  DO
  SUM := SUM + REAL(I);
  I := I + 1;
END;
```

Another concern lies in the fact that the terminal statement of a DO loop may be shared by more than one DO statement. When a shared terminal statement of a nested DO loop is a statement other than CONTINUE, the terminal statement is included only as part of the body of the innermost loop (this is consistent with the definition of the DO loop [ANS 78], pg. 11-9, in which it is specified that the terminal statement is executed only when the body of the innermost loop is executed).

Consequently the following DO loop:

```
      DO 100 I = 1,10
         DO 100 J = 1,5
            SUM = SUM + J
100      LOOP = LOOP + 1
```

is translated *as though* it were:

```
      DO 200 I = 1,10
         DO 100 J = 1,10
            SUM = SUM + J
100         LOOP = LOOP + 1
200   CONTINUE
```

where 200 is a label that does not appear in the program unit in which the statement appears.

5.5.2 Logical If statements

The logical IF statement ([ANS 78], pg. 11-3]) specifies that a single statement be executed if the value of a conditional expression is true. It has the following syntax:

```
IF ( <condition> ) <stmt>
```

where `<condition>` is a logical expression and `<stmt>` is an executable Fortran statement which is neither a DO statement, a block IF statement, an END statement, or another logical IF statement. The execution of the statement causes evaluation of the condition, after which `<stmt>` is executed if and only if the value of the condition is true.

The translation of the logical IF into Modula-2 is straightforward, and a converted logical IF statement has the following general form:

```
IF <cond> THEN <stat>
```

where `<cond>` and `<stat>` are the translation of `<condition>` and `<stmt>`, respectively.

For example, the following logical IF statement:

```
IF ( X .LT. Y) X = Y
```

is converted into the following Modula-2 if statement:

```
IF X < Y THEN X := Y END
```

5.5.3 Block IF statements

The block IF construct in Fortran ([ANS 7[illegible].pp. 11-3,11-5) allows the program to choose between one or more alternative actions. Its general form is given by the following:

```
IF ( <condition>) THEN
   <stmt-sequence>
{ ELSE IF ( <condition> ) THEN
   <stmt-sequence> }
[ ELSE
   <stmt-sequence> ]
END IF
```

where `<stmt-sequence>` is a sequence of executable statements.

Here again, the translation strategy is simple. The Modula-2 IF statement is both syntactically and semantically similar to the Fortran block IF statement, and has the following syntax:

```
   IF <condition> THEN <stmts>
{ ELSIF <stmts> }
[ ELSE  <stmts> ]
  END
```

where `<stmts>` is a (possibly empty) sequence of statements.

For example, the Fortran statement:

```
IF (X .LT. 3 .AND. Y .GT. 0) THEN
   X = X + 1
ELSE IF (Y .GT. 0) THEN
   Y = Y + 1
ELSE
   IF (Z .NE. 0) Z = Z + 1
END IF
```

is translated as follows:

```
IF (X < 3) & (Y > 0) THEN
   X := X + 1
ELSIF Y > 0 THEN
   Y := Y + 1
ELSE
   IF Z # 0 THEN  Z := Z + 1  END
END
```

5.5.4 GO TO's and Labels

A fully general Fortran to Modula-2 translation must include the elimination of **goto** statements and labels, since *standard* Modula-2 provides neither. The present translator does not eliminate goto statements from Fortran programs and as such may be seen as converting goto-less Fortran programs into Modula-2 programs. This is not to say that such a fully general translation is impossible. In fact, a number of transformation techniques have been proposed to eliminate goto's from so-called *unstructured* programs. Some of these techniques are discussed in [Fre 81], whose Fortran to Pascal translator handles the assigned and computed GOTO statements; the basis of the transformation is derived from a boolean flag algorithm proposed in [Pet 73]. Thus while in theory the translator could be extended to perform the elimination of goto's, for the purposes of this thesis no attempt to implement any such technique was made.

The translator does however process goto's and labels. In what could be perceived as an unorthodox strategy, the translator uses the TopSpeed extension of the Modula-2 language which provides facilities for goto statements and the declaration of labels. In TopSpeed, identifiers can be defined to be labels. Label identifiers can be used to label statements in Modula-2 programs, and can be referenced by GOTO statements.

In the translation, label identifiers are constructed for all labels in the Fortran program which are referenced by goto statements. These identifiers are of the form `Lab_`*n*, where n is the digit string making up the Fortran label.

5.5.5 Unconditional GO TO statements

The unconditional GO TO statement in Fortran is of the following form:

```
GO TO <label>
```

where `<label>` is the statement label of an executable statement that appears in the same program unit as the GO TO statement ([ANS 78], pg. 11-1). Unconditional GO TO statements are converted into unconditional GOTO statements of TopSpeed Modula-2 (which have the same semantics as their Fortran counterpart).

For example, consider the following Fortran code:

```
    GO TO 100
      ...
100 X = X + 1
      ...
```

The above is translated into the following TopSpeed Modula-2 code:

```
    ...
  LABEL Lab_100;        (* declaration of label *)
    ...
  GOTO Lab_100;
    ...
 Lab_100: X := X + 1;   (* labeled statement *)
    ...
```

5.5.6 Computed GO TO statement

The computed GO TO statement in Fortran is of the following form:

```
GO TO (<label> {, <label> } ) <int_expr>
```

where `<int_expr>` is an integer expression, and `<label>` is the statement label of an executable statement in the same program unit in which the computed GOTO appears. The effect of execution of the statement is that control is transferred to the statement identified by the i^{th} statement label in the list, where i is the value of the integer expression. If the value of i is not in the range $1 \leq i \leq n$, where n is the number of labels in the list of labels, then the statement has no effect, and execution resumes with the statement following the computed GOTO statement.

Computed GO TO statements are converted into Modula-2 case statements, as shown in the example below. Consider the following computed GO TO statement:

```
GO TO (100,200,300,400) I*J/2
```

Its translation is the following:

```
CASE I*J DIV 2 OF
 1: GOTO Lab_100 |
 2: GOTO Lab_200 |
 3: GOTO Lab_300 |
 4: GOTO lab_400
ELSE  (* empty statement *)
END;
```

Note that should the value of the expression be greater than the number of labels in the label list (or less than one), no action is taken and execution continues with the statement following the CASE statement.

5.5.7 Arithmetic IF statement

The arithmetic IF statement in Fortran has the following form:

```
IF  ( <expr> ) <label1> , <label2> , <label3>
```

where **<expr>** is an arithmetic expression and mbox**<label*i*>** is a statement label. When the statement is executed, the expression is evaluated, and if that value is less than zero, control is transferred to the statement labeled with **<label1>**, if the value is equal to zero, control is transferred to the statement labeled with **<label2>**, and if the value is positive, control is transferred to the statement labeled by **<label3>**.

Arithmetic IF statements are translated into Modula-2 if statements, and an additional auxiliary variable (**aux_var**) is generated to temporarily store the results of expression evaluation. The strategy is simple and is illustrated by an example. Consider the following arithmetic IF statement:

```
IF ( X+1 ) 100, 200, 300
```

Its translation is:

```
VAR aux_var: LONGREAL;
...
aux_var := LONGREAL(X+1);
IF aux_var < 0.0    THEN GOTO Lab_100
```

```
ELSIF aux_var = 0.0 THEN GOTO Lab_200
ELSE GOTO Lab_300
END
```

Note that the auxiliary variable **aux_var** is of type LONGREAL. This is done purely for the convenience of the translator so that only a single auxiliary variable need be introduced into a program unit's module for the translation of all arithmetic IF statements in that program unit. The conversion of the value of the expression to type LONGREAL does not change the semantics of the statement.

5.5.8 Assigned GO TO and ASSIGN statements

The assigned GO TO statement in Fortran has the following form:

```
GO TO <var> [ [,] ( <label> {, <label>} ) ]
```

where **<var>** is an integer variable name, and **<label>**'s denote statement labels of an executable statement that appears in the same program unit as the assigned GO TO statement. The list of labels is optional. When the assigned GO TO statement is executed, the integer variable *must* have been assigned a statement label. The ASSIGN statement assigns a label to an integer variable; it has the following general form:

```
ASSIGN <label> TO <var>
```

The translation of ASSIGN statements is not straight forward as TopSpeed Modula-2 does not support label variables (only label identifiers). The solution adopted in the translator is two-fold: firstly, the ASSIGN statement is converted into a regular assignment of an integer value (the integer interpretation of a digit string constituting a label) to the integer variable; secondly, assigned GO TO statements are constrained to have a list of statement labels (in which case Fortran requires that the assigned statement label must be present in the label list). Assigned GO TO statements can then be converted into Modula-2 case statements; case expressions are invocations of a function which takes an integer variable and a string representing the label list as arguments, and returns the position in the list of the label represented by the integer variable.

For example, consider the following Fortran statements:

```
ASSIGN 100 TO I
...
GO TO I, (200,100,300)
```

The translation is as follows:

```
I := 100; (* label treated as integer value *)
...
CASE posn(I,"200,100,300") OF
 1: GOTO Lab_200 |
 2: GOTO Lab_100 |
 3: GOTO Lab_300
END;
```

where posn returns the position in the list of labels (passed as a string) of the value currently stored in I; in this example it is 2.

5.5.9 CONTINUE statements

The CONTINUE statement, which has no effect when executed, is converted to the *empty statement* in Modula-2. For example, the following sequence of Fortran statements:

```
     GO TO 100
     ...
100  CONTINUE
     X = X + 1
```

is translated into:

```
          GOTO Lab_100
          ...
Lab_100:  ;  (* CONTINUE *)
          X := X + 1
```

5.5.10 STOP statement

The STOP statement in Fortran ([ANS 78], pg. 11-9) causes termination of execution of an executable program when executed. It has the following general form:

```
STOP [ <string> ]
```

where **<string>** is a string of not more than five digits or a character constant. At the time of termination, the value of **<string>** is *accessible*.

A STOP statement in which no string is present is converted into a HALT statement. The HALT statement in Modula-2 similarly causes program execution to cease when executed. When a string is present, the STOP statement is translated into two statements, the first of which prints out the string, and the second of which is a HALT statement.

5.5.11 PAUSE statement

The PAUSE statement in Fortran ([ANS 78], pg. 11-9) causes *resumable termination of execution* of the program when executed. It has the following general form:

```
PAUSE [ <string> ]
```

where **<string>** is a digit string or a character constant.

As Modula-2 does not provide an analogous instruction, the translator simply converts PAUSE statements into invocations of a user defined procedure called **Pause** which takes the string (if present) as argument. The definition of this procedure is system dependent. TopSpeed Modula-2 does not provide a means of resumeably ceasing program execution, and thus the execution of the Pause procedure generated by the translator simply causes the string (if present) to be output prior to HALTing program execution.

5.6 Translation of Subprograms

5.6.1 Parameter passing

Since in Fortran the type of argument passing mechanism for a given dummy argument cannot be determined without knowing the nature of the associated actual argument(s), two general approachs can be used for the generation of formal parameter lists. One involves an analysis of all possible invocations of subprograms to determine which parameters should be passed by reference and which by value. Dummy arguments associated

with expression actual arguments could be converted into value parameters, while those which become associated with variables or arrays could be converted into variable parameters. To implement such a strategy a careful global analysis of subprogram invocations is required which is complicated by the possibility of dummy procedure invocations. Another approach is to translate subprograms independently, generating formal parameters without information about corresponding actual parameters. The latter strategy is adopted by the translator.

The translator converts all dummy arguments into variable parameters except for dummy procedures and dummy arguments of statement functions. Since in mboxModula-2 actual arguments corresponding to variable formal parameters cannot be expressions, actual arguments which are expressions must be replaced by variables. Consequently, for each expression appearing as an actual argument to an external procedure an auxiliary variable (of the same type as the expression) is introduced to store the value of the expression immediately prior to invocation, and that variable is used as actual argument in place of the expression.

For example, consider the following Fortran program units:

```
REAL A,B,FUN1
...
CALL SUB1(A,FUN1(A/2,B))
...
SUBROUTINE SUB1(X,Y)
REAL X,Y
...
REAL FUNCTION FUN1(X,Y)
REAL X,Y
...
```

The subroutine `SUB1` is invoked with two arguments, the first of which is a variable (`A`) and the second is an expression (the invocation of `FUN1`). In the code generated for the invocation that is given below, two auxiliary variables (`Temp_1` and `Temp_2`) are used to temporarily store the result of the invocation of `FUN1` and the expression `A/2` respectively:

```
...
```

```
VAR A,B,Temp_1,Temp_2: REAL;
...
Temp_2 := A / 2.0;
Temp_1 := FUN1(Temp_2,B);
SUB1(A,Temp_1);
...
```

The procedure headers generated for SUB1 and FUN1 are:

```
PROCEDURE SUB1(VAR X,Y: REAL);
PROCEDURE FUN1(VAR X,Y: REAL): REAL;
```

Note that when dummy arguments are unstructured (and are not dummy procedures) the translator need only know the type of the dummy arguments in order to generate the formal parameter lists, and no global information is required.

5.6.2 Passing string arguments

Dummy arguments which are strings are uniformly converted into open array parameters. While Fortran requires that the length of a character string dummy argument be no greater than any corresponding actual argument, in the translation they are essentially of the same length (although this length may vary across procedure calls). Thus, in effect, the translator ignores the declaration of character string dummy arguments within subprograms when generating formal parameter lists. This strategy corresponds nicely with assumed size dummy character strings (which are handled by the translator) but not so well with dummy arguments which are shorter than a corresponding actual argument.

For example, consider the following:

```
PROGRAM EXAMPL
CHARACTER*20 NAME1,NAME2
CALL SUB4(NAME1,NAME2)
...
SUBROUTINE SUB4 (S1,S2)
CHARACTER*10 S1,S2
CHARACTER*20 S3
S3 = S1 // S2
...
```

Here the dummy arguments are shorter than their corresponding actual arguments, and the value stored in S3 will be the string resulting from the concatenation of the first ten characters in NAME1 and NAME2 respectively. If S1 and S2 are open array parameters in the translation of SUB4, then the procedure call Concat(Result,S1,S2) within the code for SUB4 will return the concatenation of NAME1 and NAME2 in Result.

A correct solution would involve actually extracting the first ten characters of S1 and S2 (in conformity with the declared sizes of S1 and S2 in the subprogram) and then concatenating these substrings before performing the assignment. The translator however does not utilize this strategy, and simply assumes that in the Fortran programs it makes no semantic difference that character strings are passed *in their entirety* to subprograms.

5.6.3 Passing arrays as argument

In formal parameter lists of Modula-2 procedures, type identifiers are used to declare the types of formal parameters (except for open array parameters). When parameters are structured objects, the program must define type identifiers for their declaration. Consequently, the translator generates type identifiers for all arrays that appear as an actual or dummy arguments in the Fortran program. Since Modula-2 generally requires that actual and formal parameters be defined using the same type identifiers, array arguments pose somewhat of a problem, because type identifiers for arrays specify their index type which statically define the upper and lower bounds of subscripts (which need not be the same in actual and dummy arguments in Fortran).

The strategy adopted by the translator to deal with this situation is simple (alternative strategies are discussed in the conclusion); actual argument arrays are required to be identical both in size and in the range of subscripting values to all dummy arguments with which they become associated. This effectively permits the same type identifier to be used in the declaration of both actual and dummy arguments. By the same token, the restriction prohibits the use of adjustable and assumed size arrays.

In Fortran, when an array of character strings is supplied as actual argument, the only

restriction on the corresponding dummy argument is that the *total* amount of storage implied by the dummy argument does not exceed that of the actual argument. The length of the string elements of the dummy array may differ from that of the actual argument array's elements. This situation is problematic, since while open array parameters can have more than one dimension, only the first dimension is flexible. As a result, the translator requires that the type of character string array actual and dummy arguments be the same, and open array parameters are not used in the passing of these objects.

The identifiers used as type identifiers in the translation are constructed so as to reflect the characteristics (ie., the index range and the element type) of array structures. For instance, the type identifier for an array declared in the Fortran program with the following type statement:

```
INTEGER A(3:57)
```

would be `Int_array_3_57`. In this way, the type identifier generated for every integer array with indices ranging from 3 to 57 will be the same thereby allowing their generation (but not their definition) to be done locally. As the type identifiers of actual and formal parameters must have the same defining occurrence, all generated type identifiers are collected together and defined (exactly once) in a module called `Types`, and modules which declare objects of those types which are formal or actual parameters import those identifiers from that module. An example is the following:

```
PROGRAM P
INTEGER NUM(10)
...
CALL SUB1(NUM)
...
END
SUBROUTINE SUB1(L)
DIMENSION L(10)
...
END
```

Here, both the actual and dummy argument for `SUB1` are integer arrays with ten elements whose subscripts range from 1 to 10. Its translation is outlined in the following:

```
MODULE P;
FROM Types IMPORT Int_array_1_10;
VAR NUM : Int_array_1_10;
...
SUB1(NUM);
...
END P.

DEFINITION MODULE SUB1_mod;
FROM Types IMPORT Int_array_1_10;
PROCEDURE SUB1(VAR L:Int_array_1_10);
...
END SUB1.

DEFINITION MODULE Types;
TYPE Int_array_1_10: ARRAY [1..10] OF LONGINT;
...
```

One further restriction on array arguments is introduced, namely, when an array element is used as actual argument, the corresponding dummy argument must be of the same type. Since the translator assumes that all arrays in the Fortran program are one-dimensional, this means that the dummy argument must be a variable of the same type as the array element (or a string). Consequently, only the entire array can be passed to a procedure. Again, alternatives are discussed in the conclusion.

5.6.4 Subroutines

For each Fortran subroutine the translator generates a module containing the definition of an analogous Modula-2 procedure. RETURN statements in a subroutine are converted into Modula-2 RETURN statements. In Fortran, the CALL statement is used to invoke subroutines. CALL statements are converted to Modula-2 procedure invocations, possibly preceded by assignment statements for evaluation and temporary storage of the values of expression actual arguments.

Alternate return specification

In Fortran when execution of a subprogram terminates, control is normally returned to the executable statement following the CALL statement with which the subroutine was invoked. This convention can, however, be circumvented using *alternate return specifiers* ([ANS 78], pp. 15-11, 15-19) which are 'passed' as arguments to the subroutine. An alternate return specifier is an actual argument which has the syntax `*<label>`, where `<label>` is a statement label of an executable statement in the invoking program unit. Dummy arguments which correspond to alternate return specifiers are asterisks (*).

A subroutine which takes alternate return specifier(s) as arguments may optionally specify that control be returned to one of the labels supplied as argument by including an integer expression in RETURN statements. The integer expression identifies which (if any) of the alternate returns is to be chosen. The following example demonstrates the use of alternate return specifiers:

```
SUBROUTINE SUB (A,*,B,*)
INTEGER A,B
...
IF (A .LT. 0) THEN
   RETURN 1
ELSE IF (B .LT. 0) THEN
   RETURN 2
ELSE
...
END
...
PROGRAM CALLER
...
CALL SUB(A,*100,B,*200)
...
```

If SUB is invoked from CALLER, then if the statement RETURN 1 is executed within SUB control returns to the statement labeled by 100 (ie., the alternate return associated with the first asterisk dummy argument) in CALLER; if RETURN 2 is executed, control returns to the statement labeled by 200. If the value of the integer expression used in a RETURN

statement is less than one or greater than the number of asterisks in the dummy argument list, then the effect of the RETURN statement is the same as if it were used without an integer expression.

The translator requires that any integer expression appearing in a RETURN statement be an integer literal. The strategy for translating alternate return specification is to replace alternate return specifiers by boolean variables which are initialized to FALSE before procedure invocations. Inside the procedure, RETURN statements which contain an integer value are converted into two statements, an assignment statement which which sets the appropriate boolean variable followed by a RETURN statement.

For example, the invocation above would be converted into:

```
Lab_100_flag := FALSE;
Lab_200_flag := FALSE;
SUB(A,Lab_100_flag,B,Lab_200_flag);
IF Lab_100_flag THEN GOTO Lab_100 END;
IF Lab_200_flag THEN GOTO Lab_200 END;
```

while the procedure definition generated for SUB would be:

```
PROCEDURE SUB(VAR A:REAL; VAR flag1:BOOLEAN;
              VAR B:REAL; VAR flag2:BOOLEAN);
BEGIN
  IF A < 0 THEN
     flag1 := TRUE;
     RETURN
  ELSIF B < 0 THEN
     flag2 := TRUE;
     RETURN
  ELSE
  ...
END SUB;
```

The restriction that expressions in RETURN statements must be integer literals is imposed to simplify the generation of assignments to the boolean variables. An alternative is given in the concluding chapter.

5.6.5 External functions

In a Fortran external function, the name of the function must appear as a variable name. During execution of the function, this variable must become defined, and once defined, may be referenced or redefined. The value returned by the function is the value of this variable when a RETURN or END statement is executed. For example, consider the following function definition:

```
   INTEGER FUNCTION ABSDIF(M,N)
* poorly written for the sake of example!
   INTEGER M,N
   ABSDIF = 0
   IF (M .EQ. N) THEN
      RETURN
   ELSE IF (M .LT. N) THEN
      ABSDIF = N - M
   ELSE
      ABSDIF = M - N
   END IF
   END
```

The value returned by the function is the value of variable ABSDIF when either the END statement or the RETURN statement is executed.

In the Modula-2 function definitions, a local variable (with the same name as the function identifier) is declared, and corresponds to the function variable of Fortran functions. RETURN statements in Fortran functions are converted into RETURN statements in Modula-2 in which that variable used to indicate the value to be returned, and the keyword END in the Modula-2 function is always preceded by a RETURN statement of this form.

For example, the function above would be translated into the following:

```
PROCEDURE ABSDIF(VAR M,N: LONGINT): LONGINT;
VAR ABSDIF: LONGINT;
BEGIN
  ABSDIF := 0;
  IF M = N THEN
     RETURN ABSDIF
```

```
  ELSIF M < N THEN
     ABSDIF := N - M
  ELSE
     ABSDIF := M - N
  END;
  RETURN ABSDIF
END ABSDIF;
```

Notice that the scope of the function identifier `ABSDIF` does not include the body of the function.

5.7 Input and output statements

In light of the fact that any strategy for the translation of Fortran input and output statements into Modula-2 is necessarily non-standard (since Modula-2 provides no standard procedures or statements for performing I/O), the goal of the translator is to provide a suitable abstraction of the details of input and output handling. While the `FIO` module in the TopSpeed system (chapter 4) provides routines which permit sufficiently low-level control over input and output to translate I/O in Fortran, the task of developing implementation level strategies for the translation of Fortran I/O into TopSpeed Modula-2 was considered beyond the scope of this thesis.

The abstraction is provided by translating input and output statements into sequences of invocations of procedures which are imported from a module called `FinOut`. Procedures defined in `FinOut` must in turn make use of the I/O facilities provided by the Modula-2 system in the target environment. As such, `FinOut` controls all aspects of I/O in generated programs; only procedures defined in `FinOut` are used for I/O operations, and `FinOut` maintains its own private variables which are needed to simulate record-oriented I/O, direct access to records, etc. Since specific implementations of `FinOut` will vary depending on the particular Modula-2 system employed, this section describes a general approach to the translation of input and output statements and no attempt is made to exhaustively cover the remarkable variety of I/O facilities of Fortran.

5.7.1 The READ statement

At the lowest level, formatted input in Fortran is the process of accessing data in the form of a string of characters (a record) from an external device and transmitting them to a buffer (normally inaccessible to the Fortran program). This character string is then converted into the form required by the internal representation (integer, real, logical, character, etc.). Format descriptions specify the positions of the *fields* of records and the character(s) in each field collectively represent in some way a single value of a particular type specified in the description.

For example the format description:

```
(I2, F6.2, I4)
```

describes the arrangement of three fields within a record. The first two characters in the record represent an integer value, the next six represent a real number, and the ninth through twelfth represent another integer value. Thus a record consisting of the characters `123456789012` represents, according to the above format description, the values 12, 3456.78, and 9012 (in that order).

Format descriptions in conjunction with READ statements establish an association between the values represented in record fields and variables appearing in READ statements. For instance, consider the following READ[25] statement:

```
      READ 20, I,X,N
20    FORMAT (I2, F6.2, I4)
```

If this statement was executed and the "next" record to be accessed consisted of the characters `123456789012`, then the values 12, 3456.78, and 9012 would be assigned to the variables `I`, `X`, and `N`, respectively.

[25] Note that the READ statement is in its so-called "short-form", a form used when the only item of control information is a format description. The fully general form of the READ statement includes a *control list* specifying an external device, the format of the data on the device, and (optionally) additional information about, for example, action to be taken when the end of the input data is reached.

Modula-2 library procedures for reading numeric and boolean values are generally stream oriented and require some kind of delimiting character in the input stream. Subsequently there is no "easy" way to specify that, for instance, the next two characters in the input stream are to be interpreted as an integer value. Procedures for input of character strings similarly make use of delimiting characters, although the length of the string variable supplied as argument may implicitly specify a maximum number of characters to be read in a given invocation (ie., reading of characters into a string variable may stop before a delimiting character is encountered if the string variable has been "filled up"). Moreover, these procedures only read a single value at a time.

Clearly, these higher level input procedures cannot be used to to implement or simulate the kind of record-oriented input that is needed in the translation of Fortran programs. For this reason, a module such as `FinOut` is needed to provide "format-driven" input in which fixed size representations of values (corresponding to record fields) can be accessed and appropriately converted into values according to format descriptions. To accomplish this, `FinOut` must in some way be able to represent format descriptions derived from Fortran programs. Once represented and stored within `FinOut`, procedures for reading values can reference such a description to determine both the length of the datum to be read in a given invocation (ie., the number of characters used in the representation of the datum) and the type of conversion that needs to be performed on the character input. Subsequently, these procedures can access data by performing character-by-character input, and convert data from string form by either using one of the procedures provided by the Modula-2 system for converting between strings and other types (most Modula-2 systems provide a wide variety of such conversion procedures) or a user-written procedure.

For further clarification consider the following sequence of procedure calls, which is generated by the translator from the Fortran READ statement above:

```
Format ("I2, F6.2, I4");
Read_Int(I);
Read_Rl(X);
Read_Int(N);
```

All of the procedures invoked are imported from FinOut. The procedure Format is used to initialize variable(s) used to store the format description supplied as argument, as well as to initialize a pointer of the 'next applicable format descriptor'. Procedures Read_Int and Read_Rl read a single value (of integer and real type, respectively) into the variable supplied as argument. One such read procedure is assumed to be defined in FinOut for input of each of the types represented in Fortran programs. These procedures reference the format descriptor pointed to by the 'next format descriptor' pointer to determine how many characters are to be read from the input stream and how to convert the resulting string[26]. They are also responsible for updating the pointer after the read operation. Naturally, this updating can be complicated by the existence of repeat counts associated with descriptors, and by the possibility of "re-scanning" part or all of the format description when the number of variables in a READ statement is greater than the number of descriptors, but this simply implies that more than a pair of variables are required to control such format-driven input.

The translation of READ statements with array variables can be accomplished with the same strategy, using iteration. For example, consider the following Fortran code:

```
      INTEGER NUM1(10), NUM2(10)
       ...
      READ 30, X,NUM1, (NUM2(I),I=M,N)
30    FORMAT (F5.1, (I3))
```

Its translation is the following:

```
     FROM FinOut IMPORT Format;
     FROM FinOut IMPORT Read_Rl;
     FROM FinOut IMPORT Read_Int;
VAR  NUM1: ARRAY [1..10] OF LONGINT;
     NUM2: ARRAY [1..10] OF LONGINT;
     ...
     Format("F5.1, (I3)");
     Read_Rl(X);
```

[26]In practice, more than one descriptor may come to bear on the input of a single value, such as when an X or slash descriptor appears in a format description.

```
FOR i := 1 TO 10 DO
  Read_Int(NUM1[i])
END;
FOR I := M TO N DO
  Read_Int(NUM2[INTEGER(I)]);
END;
```

Note that the same framework could be used for handling *list directed formatting* (input 'controlled' by the input list, in which separators are used to delimit values represented in records). For instance, the statement

```
READ *, A,B
```

is converted into

```
Format("*");
Read_R1(A);
Read_R1(B);
```

In this case, the procedure `Read_R1` will detect the use of implicit formatting and act accordingly (including the skipping over of separator characters).

5.7.2 The WRITE and PRINT statements

The same general strategy can be employed in the treatment of formatted output in Fortran. Generated Modula-2 programs will import from `FinOut` procedures for outputing values of each (printable) Fortran type. When an output operation is being performed, `FinOut` maintains a representation of the format description which applies, and a pointer to the 'next applicable descriptor' in that description. The implementation of the output procedures in `FinOut` is simplified by typical feature of Modula-2 library procedures for output which require the size of the field on which values are to be printed to be specified in one of the arguments. Some additional care will have to be taken within these procedures to account for any control characters specified in the format description. Moreover, these output procedures must be able to detect and react to the existence of *apostrophe editing* (ie., when character constants appear in format descriptions for output).

For example, consider the following:

```
      PRINT 40, X
40    FORMAT ('0', 'The value of X is', F6.2)
```

Its translation is the following:

```
Format("'0', 'The value of X is', F6.2");
Write_R1(X)
```

In this example, the procedure `Write_R1` must first cause the control character and the string constant to be output prior to printing the value of `X` in the format prescribed.

The PRINT statement in Fortran is the "short form" of the more general output statement, the WRITE statement, which includes a control list. The WRITE statement, as well as the general form of the READ statement, can be translated only if additional Fortran I/O statements are handled (such as the OPEN and CLOSE statements - see following section). Since these additional statements are not handled by the translator, the WRITE statement is not translated although the underlying strategy for the translation of the WRITE statement would be the same as that described for handling the PRINT statement.

5.7.3 Other I/O statements

The translator does not handle any I/O statements other than the READ statement (without a control list) and PRINT statements[27]. Again, the motivation for this decision is based on the necessarily non-standard nature of any translation strategy for Fortran I/O. But it should be stressed that such a translation is not necessarily impossible for any given Modula-2 system. For instance, the Fortran I/O statements OPEN and CLOSE have direct counterparts in TopSpeed Modula-2, namely the `Open` and `Close` procedures in module `FIO`, and differences between them are relatively minor. TopSpeed also provides procedures for file positioning (`GetPos` and `Seek`) which could be used in the implementation of the Fortran statements BACKSPACE, REWIND, INQUIRE, and ENDFILE

[27] Consequently, formatted internal data transfer is not handled. Alternatives are discussed in the concluding chapter.

without great difficulty (the major difference between the Fortran statements and the Modula-2 procedures are that the latter are byte rather than record oriented).

Chapter 6

Implementation

6.1 The attribute grammar

In this section an overview of the AG describing the translation is given. A single tree is used to represent the entire executable Fortran program, and the use of attributes is described largely in terms of tree structures.

Attributes in the grammar can be characterized by their use. *Environmental attributes* are attributes used to generate information about the environment of a program. Attributes of this type typically derive their values from the definitions and declarations (implicit or explicit) of entities in the program (ie., most environmental information is derived from specification statements), and their values are used in the construction of Modula-2 definitions and declarations.

We can distinguish from environmental attributes those which are used primarily in the generation of code for executable statements. These attributes derive their values from the *use* of entities in the executable statements of the program (in conjunction with environmental attribute values).

6.1.1 Environmental attributes

As the global variable concept does not exist in Fortran, most environmental information for a given program unit is derived from the subtree representing the program unit.

The main attributes used for the representation of environmental information within a given program unit are **env_in**, **env_out**, and **env**. The first two (inherited and synthesized attributes, respectively) are used to *accumulate* the environment of a program unit[1]. **env** is used to pass the complete description of the environment synthesized by **env_out** at the root of the subtree for a program unit back into the nodes in the subtree; more specifically, **env** is inherited by *some* internal nodes in the subtree, for the most part, nodes in subtrees representing the executable statements[2].

The domain of these attributes is a list of *entity descriptions.* Each entity description is a structure consisting of fields for: the name of the entity (a symbol), its type (integer, real, character string, untyped, etc.), a description of the object it represents (the nature of the description depends on whether it is a constant, variable, array, dummy variable, dummy array, statement function, external function, or dummy procedure, etc.), its membership in a common block or equivalence class, and whether or not it is to be 'saved'.

As Fortran does not require the declaration of variables before their appearance in executable (and DATA) statements, **env_in** and **env_out** are used to gather the environment of a program unit from both the specification statements and the executable statements in a program unit.

The environment is updated each time a new entity is declared (explicitly or implicitly) or additional information about an entity is specified (such as when a previously typed

[1]Often, a pair of attributes - one inherited and one synthesized - are used *in tandem* to accumulate information of the same type in a tree or subtree. Typically, the value of the inherited attribute at a node represents the accumulated information *prior to* encountering the structure represented by the node, while the value of the corresponding synthesized attribute at the node represents the accumulated information *after* the structure represented by the node has been encountered. Attributes **env_in** and **env_out** are such a pair. In the AG, the naming convention for such pairs is to suffix the inherited attribute name by '_in' and the synthesized attribute name by '_out'.

[2]Subtrees representing the expressions in statement function definitions inherit the environment of the program unit modified to include only those entities which do not have the same name as any of its dummy arguments, and only those statement functions which precede its definition.

entity is dimensioned through its appearance in a DIMENSION statement). Additional entities may be added to the environment which do not correspond to Fortran entities when executable statements whose translation involves the addition of auxiliary variables (such as the DO statement) are encountered.

Several attributes are used to support the computation of the environment. Attributes `imp_in` and `imp_out` are used to gather and represent any implicit typing rules specified in a program unit. These rules are used to determine the types of unexplicitly typed entities. All subtrees for program units inherit the Fortran default implicit typing rules, some or all of which may be overridden by typing rules specified in IMPLICIT statements.

The attributes `lab_in` and `lab_out` are used to generate a list of all labels of executable statements in a program unit.

Attributes representing entity association

There are also attributes which represent more detailed information about entities than that represented in `env`. In particular, attributes are used to represent the association of entities in a program unit with common blocks and/or equivalence classes. While the definition in `env` of an entity which is in a common block or equivalence class includes a reference to the particular block or class with which it is associated, these attributes describe their relative positions in the storage sequence with which they are associated. The attributes `cblks_in` and `cblks_out` accumulate descriptions of the common blocks referenced in a program unit, while the attributes `equiv_classes_in` and `equiv_classes_out` are used to accumulate and represent the equivalence classes in a program unit.

This abstraction of the particulars of the association of entities from the general description of the environment in `env` is done for two reasons: firstly, the relative positions in a storage sequence of associated entities is not, strictly speaking, necessary for the generation of code for executable statements (ie., it is sufficient to know *which* block or equivalence class an entity is associated with in order to generate references to it in the Modula-2 program). Secondly, by centralizing such information in these attributes, the

computation of the relative positions of associated entities is facilitated. Moreover, common blocks are global objects, and information about their declaration within a program unit has to be passed 'outside' of the subtrees for program units (see below).

The computation of the values of `cblks` and `equiv_classes`[3] is done in two distinct phases. The first phase consists of extracting the *raw* information provided in specification statements. A list of all referenced common blocks is generated from the COMMON statements, along with the names, characteristics (ie., type and structure), and order of appearance of the entities in each block. From the EQUIVALENCE statements a list is generated each element of which corresponds to a single EQUIVALENCE statement (ie., a list of variable names, array names, and array element names which appear in the same equivalence statement - we call such a list an *equivalence list*).

The second phase consists of three steps. First, for each equivalence list the relative offsets of each entity from the start of the storage sequence implied by the list are computed. Second, disjoint equivalence classes are formed by 'merging' equivalence lists which share entities, generally resulting in the modification of the relative offsets of some of the entities. Finally, equivalence classes which contain at least one entity which is a member of a common block are 'merged' with their associated common block, possibly resulting in further modification of the value of the offsets of entities in the class. The final value of the offset computed for a given entity in an equivalence class is used in the construction of the record field associated with it in the Modula-2 program.

Subtrees for program units are themselves composed of three main subtrees representing the declarations, the DATA statements, and the executable statements of the program unit. The first phase described above is performed during the first visits to the nodes of the subtree representing the declarations in the program unit. The computations corresponding to the second phase are done after the first visit to the nodes of the declaration subtree. Once performed, the environment synthesized by that subtree is updated to re-

[3]When we use these attribute names without the suffixes _in and _out, the meaning is the value accumulated by the corresponding pair of attributes in a subtree.

flect the actual membership of entities in common blocks or equivalence classes (which can now be associated with distinct integer values - see section 5.4.4) so that correct references to such entities can be constructed for the Modula-2 code.

Attributes inherited by program units

While most information required to represent the environment of a program unit is derived from the subtree representing the unit itself, some global information is needed.

Most importantly, program units may reference external procedures defined elsewhere in the executable program. Subtrees for program units inherit the attribute **externals** which lists the external procedures in the executable program; these external procedure descriptions are incorporated into the environment represented by **env**. Moreover, this information provides the basis for determining whether a given function invocation (which is not a statement function invocation) is an intrinsic function invocation or an external procedure reference[4]. Naturally, all external procedure references in CALL statements must use one of the external procedures described in **externals**.

Additionally, program units inherit an attribute which contains an integer value which is uniquely associated with the program unit. This value is used in the construction of field identifiers of records representing common blocks in the Modula-2 code (see section 5.4.3)

Note that while common blocks are global objects, the nature of their definition outside of a given program unit is not required within that program unit, since the names used to refer to locations within common blocks are local to the program units that refer to them.

Attributes synthesized by program units for global use

While most environmental attribute values are only required locally, some are needed outside of the subtree representing a given program unit. Such attributes are synthesized

[4]This is a kind of *shortcut* taken by the translator. A more robust version could check the names of functions invoked in a program unit which do not appear in externals against a list of intrinsic function names.

by the root of program unit subtrees making them available to nodes *higher up* in the tree of the entire program.

For one, the value of the attribute which represents the constituents of a common block in one program unit needs to be somehow combined with the values of the same attribute representing common blocks in other program units in order to construct the definition of module `COMMON` in the Modula-2 program. Consequently, program unit subtrees synthesize the ('processed') value of `cblks`, and the *combining* of the values synthesized by all program units is done at the root of the tree for the executable program.

As type identifiers needed in the Modula-2 program are declared outside of the modules containing the code for program units, subtrees for program units synthesize (using attributes called `types_in` and `types_out`) a list of type descriptions for types of structured entities (arrays), procedures, and dummy procedures which are used as arguments within the program unit. Higher up in the tree for the executable program, the *union* of these type descriptions is formed, and the resulting list is used to generate the definition of module `Types` in the Modula-2 program.

Finally, each program unit subtrees synthesize an attribute which describes the program unit itself. The value of this attribute is the name and type (subroutine or function) of the program unit *if it is an external procedure* (otherwise it has a 'null' value). This information is used to synthesize a list of external procedures in the executable program at the root of the tree for the executable program, and the synthesized list is ultimately inherited by subtrees for program units in `externals`.

6.1.2 Code generation

All Modula-2 source code is represented in ALADIN as a list of *tokens* (a user-defined type in the AG specification). A token (implemented as a discriminated union type value) can be a symbol, a string, an integer literal, a real literal, a qualified name, etc. The actual output of Modula-2 code is performed *via* ALADIN external functions which convert the ALADIN representation of tokens into a form suitable for output by the Pascal functions

(normally a sequence of characters). Typically, this involves accessing the translator's symbol table.

Definitions and declarations

For the most part, the complete environment provides sufficient information for the generation of the definitions and declarations in the Modula-2 code. Definitions and declarations which are to appear in modules corresponding to program units are generated using the values of the environmental attributes synthesized at the root of the subtree representing it; the value of `env` is used to construct the definitions of constants, unassociated variables and arrays, and formal parameter lists, as well as the import statements for all imported procedures and translator provided types (such as `Logical` and `Complex`). The value of `equiv_classes_out` is used to construct record variable declarations for the equivalence classes of the unit. The value of `cblks_out` is used to generate import statements for common blocks, `types_out` is used to generate import statements for types imported from `Types`, and `labs_out` is used to construct label definitions.

The only definitions local to the module for a program unit which are not generated from the environmental attributes are the definitions of functions and procedures constructed for statement functions and alternate returns, or import statements for procedures and/or functions used in the translation of executable statements. Generation of these definitions is performed during code generation for executable statements (see below).

The definition of the objects in module `COMMON` is based on the 'combined' values of `cblks` from each program unit which references common blocks.

Executable statements

Code generation for executable statements (and DATA statements) is performed after the environmental information has been synthesized. The attribute `code` is primarily responsible for synthesizing the code. Code for executable statements is generated on a

statement-by-statement basis. For instance, the code generated for an IF statement is synthesized by **code** at the root of the subtree representing the IF statement (naturally, subtrees of the IF stmt subtree will synthesize code for the condition(s) and statements in the body of the IF statement). Typically, the value of **code** is computed during the final visit to the nodes with which it is associated.

A few attributes are used to help compute the value of **code**, and the attributes involved depend upon the context of the occurrence of **code**. Only a few examples are given here.

An attribute called **type** synthesizes the type of expressions. For example, in the generation of code for an assignment statement, the value of **type** at the root of the expression subtree is used to determine whether the corresponding Modula-2 expression needs to be converted to the type of the target variable in the assignment statement. The same attribute is used in the subtrees of the expression to determine whether any type conversion of operands is necessary within that expression.

When external procedures are invoked with expressions as arguments, additional assignment statements are needed to store the values of these expressions in auxiliary variables (added to the environment during the first pass) prior to invocation of the corresponding Modula-2 procedures. Consequently, an attribute **aux_assns** is associated with all external procedure invocations, and it synthesizes code for such assignments, if necessary. As external function invocations are themselves expressions, this attribute is associated with all expressions. In order to avoid unneeded generation of such auxiliary assignment statement code (such as when an expression appears in any context other than as an argument), expressions inherit yet another attribute which indicates whether it represents an actual argument of an external procedure or not.

Code for the definitions of statement functions also must be constructed *after* the complete environment has been synthesized (as expressions in statement functions may reference entities which are not dummy arguments). Attributes called **fun_defs_in** and **fun_defs_out** are used to accumulate a list containing descriptions of the statement functions of a program unit, and the synthesized list is used to generate the necessary

Modula-2 definitions.

6.2 Scanner preprocessing

A few minor transformations on the Fortran source code are performed by the scanner. These are done to avoid one of three kinds of problems: the overloading of syntactic constructs, the weaknesses of the LALR(1) parser generator, and the difficulty of expressing certain syntactic features of Fortran using context free grammars. These transformations are described below.

6.2.1 Overloading of syntactic constructs

This situation arises when semantically different constructs share the same syntax. To handle such cases, two approaches are possible[5]: either a single production can be used for 'both' constructs (in which case the semantic rules associated with the production distinguish between the two), or different productions can be used for each (in which case the parser - with the aid of the scanner - must somehow distinguish between the two cases).

Examples of such conflicts exist in Fortran. For example, array element assignment statements and statement function definitions share the same syntactic structure. In the AG for the translator, a single production is used for both and the semantic rules determine the semantics of the structure on the basis of the definitions in the environment. The drawback of this approach is that the associated semantic rules become cumbersome; a number of attribute occurrences are defined which really only play a role in one case or the other (for instance, the attribute synthesizing the definition of a statement function must be defined even for the case when the construct is an array element assignment statement). Most attribute expressions in semantic rules had to be made into conditionals in which the value chosen depended on the description of the defined or referenced entity in the

[5] An alternative strategy for AG based translation is discussed in [Far 89].

environment. Moreover, this *overloading* is propagated through the productions; just as a structure such as A(X,Y,Z)=X+Y+Z is either a statement function definition or an array element assignment, so is (X,Y,Z) either a dummy argument list or a list of subscript expressions, and X either a dummy argument or an expression, and so on. Worse still, additional attributes had to be introduced into the grammar to propagate the context derived by the semantic rules further down into the tree (so that, for instance, the correct code for X, Y, and Z is generated - either formal parameter declarations or expressions).

The translator employs a different strategy for handling expressions and assignment statements (in which the target of the assignment is a variable). Just as an expression consisting of a single name can be either an arithmetic expression, a logical expression, or a character string expression, so can an assignment statement of the form <name> = <name> be either an arithmetic assignment statement, a logical assignment statement, or a character assignment statement. As the semantics of each are different (and consequently the semantic rules associated with them), the translator defines separate productions for each.

In order to accomplish this, however, the parser must be able to distinguish between the names of entities of the different types to determine which production has been applied. Since parsing is performed before the execution of semantic actions (ie., the evaluation of attributes), the additional information needs to be supplied by the scanner. In the translator, the scanner recognizes logical and character string names as different kinds of tokens from all other symbols. To facilitate the scanner's task, the translator requires that both LOGICAL and CHARACTER string names be explicitly typed[6]. In its implementation, the scanner recognizes LOGICAL and CHARACTER type statements, and 'remembers'[7] names which have appeared in them, so that when they are encountered anywhere in the rest of the program unit the appropriate token type is returned to the parser.

[6]This restriction could be removed by the implementation of a more powerful scanner which recognizes implicit typing rule specifications.

[7]this had to be implemented by introducing an auxiliary symbol table into the translator, since the translators regular symbol table was particularly uncooperative in associating different token types with symbols other than those provided by the system.

6.2.2 Parser limitations

A couple of syntactic features of Fortran could not be handled by LALR(1) parsing. Problems arose due to the significance of textual positioning of statements in Fortran programs (there is never any ambiguity in Fortran about where one statement ends and another begins because they will always be on textually different lines).

For instance, the tokens ELSE and IF appearing in succession can signify one of two things: either the start of a so-called ELSE IF statement (if they appear on the same line) or the start of an ELSE statement the first statement within which is an IF statement. This creates an ambiguity for LALR(1) parsers. To resolve the conflict, the scanner converts ELSE IF statements into a single keyword ELSEIF, and that keyword is used in the production rule for block IF statements.

A conflict also arises when DATA statements are not somehow explicitly terminated. Consequently the scanner inserts a special marker at the end of DATA statements, and this marker appears in the production for DATA statements.

6.2.3 Shared terminal statements in DO loops

Another problem was to express the *sharing* of a terminal statement of a DO loop by more than one loop. An example is the following:

```
      DO 20 I = 1,10,2
         DO 20 J = 1, I
            SUM = SUM + J
20    CONTINUE
```

Such structures are awkward to express using context free grammars. To avoid this problem, the scanner converts DO loops into the following form:

```
    DO  <label>, <ident> = <expr> , <expr> [ , <expr> ]
        { <statement> }
<label>  <statement>
    ENDDO
```

Thus each DO statement is paired with the keyword ENDDO. So the example above would be transformed by the scanner into

```
      DO 200, I = 1,10,2
         DO 200, J = 1, I
           SUM = SUM + J
200        CONTINUE
         ENDDO
      ENDDO
```

The scanner uses a 'label stack' to store labels of terminal statements. When the start of a DO loop is detected, it pushes the label of its terminal statement onto this stack. Whenever a label appears in the code which follows, it checks to see if it is the same as the one currently on the top of the stack; if it is, the stack is popped and the keyword 'ENDDO' is put into the token stream after the statement labeled by it. This process is repeated until the label on the top of the stack is no longer the same as the terminal statement label.

6.3 The translator

The translator consists of about 40,000 lines of Pascal code generated by the GAG system from an ALADIN attribute grammar specification of approximately 7000 lines (roughly one third of which are ALADIN function definitions). The context free grammar contains 250 production rules with 110 nonterminal symbols. The attribute grammar contains 1200 attribute rules, 100 of which are semantic conditions, and defines 400 attributes.

The translator currently runs on a MicroVAX III under an Ultrix operating system. A shell procedure provides the interface between the user and the GAG generated program. Input to the translator consists of any number of files each containing source code for any number of Fortran program units, but only one main program unit may appear in the source files. The output consists of a number of files each containing a Modula-2 module definition (the definition and implementation modules of a given separately

compiled module are stored in separate files). The output files are named in accordance with TopSpeed Modula-2 file naming conventions.

It should be noted that the translator is not intended to be a full syntax and semantic analyzer for Fortran-77 programs. A number of semantic checks are performed on the source code by means of semantic conditions, however, this checking is not complete (although the translator *could* be extended to exhaustively perform such semantic checks). A complete syntactic check, in accordance with the context free productions, is performed by the embedded parser, and messages are produced for syntax errors. Messages are also output upon detection of numerous semantic errors and violations of translator imposed restrictions on Fortran programs. A complete listing of the restrictions imposed by the translator is included in chapter 8.

Chapter 7

Examples

In this chapter we present a couple of short but complete examples. The examples are intended to provide some sense of the *flavour* of the translator-generated programs. The first consists of a number of program units which manipulate entities in a common block, while the second consists of a collection of program units which communicate *via* parameter passing mechanisms.

7.1 Example 1

Fortran-77 source code:

```
      PROGRAM EXAM1
      COMMON /BLK/ N,A(200)
      LOGICAL ASCEND
      LENGTH  = N
      IF (ASCEND(LENGTH)) CALL REVERS
      END

      LOGICAL FUNCTION ASCEND(LN)
* returns true if first LN elements in A (in BLK)
* are in ascending order
      COMMON /BLK/ N,A(200)
      ASCEND = .TRUE.
      I = 1
 20   IF (I .GE. LN) THEN
```

```
         RETURN
      ELSE IF (A(I) .GE. A(I+1)) THEN
          ASCEND = .FALSE.
          RETURN
      END IF
      I = I + 1
      GO TO 20
      END

      SUBROUTINE REVERS
      COMMON /BLK/ N, Q(200)
      MIDDLE = N / 2
      DO 10 I =1, MIDDLE
         CALL SWAP(Q(I), Q(N+1-I))
10    CONTINUE
      END

      SUBROUTINE SWAP(A,B)
      TEMP = A
      A = B
      B = TEMP
      END

      BLOCK DATA INITBLK
      COMMON /BLK/ N,A(200)
      DATA N/5/ A(1),A(2),A(3),A(4),A(5)/5,4,3,2,1/
      END
```

Generated Modula-2 code:

```
MODULE EXAM1;(* main program module *)
FROM COMMON IMPORT BLK;
FROM ASCEND_O IMPORT ASCEND;
FROM REVERS_O IMPORT REVERS;
FROM Types IMPORT Logical;
VAR
  LENGTH: LONGINT;
BEGIN   (* module body *)
LENGTH := BLK.N_1;
IF BOOLEAN(ASCEND(LENGTH)) THEN
  REVERS;
END; (*  IF  *)
```

```
END EXAM1.  (* main program module *)

DEFINITION MODULE ASCEND_0;
FROM COMMON IMPORT BLK;
FROM Types IMPORT Logical;
PROCEDURE ASCEND (VAR LN: LONGINT): Logical;
END ASCEND_0.

IMPLEMENTATION MODULE ASCEND_0;
PROCEDURE ASCEND (VAR LN: LONGINT): Logical;
LABEL Lab_20;
VAR
  ASCEND: Logical;
  I: LONGINT;
BEGIN  (* procedure body *)
ASCEND:= Logical(TRUE);
I := 1;
Lab_20: IF (I>=LN)THEN
RETURN ASCEND;
ELSIF (BLK.A_2[INTEGER(I)]>=BLK.A_2[INTEGER(I+1)]) THEN
ASCEND:= Logical(FALSE);
RETURN ASCEND;
END;
I := I+1;
GOTO Lab_20;
RETURN ASCEND;
END ASCEND;  (* procedure body *)
END ASCEND_0.

DEFINITION MODULE REVERS_0;
FROM SWAP_0 IMPORT SWAP;
FROM COMMON IMPORT BLK;
PROCEDURE REVERS();
END REVERS_0.

IMPLEMENTATION MODULE REVERS_0;
PROCEDURE REVERS();
LABEL Lab_10;
VAR
  MIDDLE: LONGINT;
  I: LONGINT;
(* translator generated variables *)
```

```
  Start_1: LONGINT;
  Limit_1: LONGINT;
  IterCount_1: LONGINT;
  IterCtrl_1: LONGINT;
BEGIN  (* procedure body *)
MIDDLE := BLK.N_3 DIV 2;
(* DO I = ...  *)
Start_1 := 1;
Limit_1 := MIDDLE;
I := Start_1;
IterCount_1 := Limit_1 - Start_1 + 1;
FOR IterCtrl_1 := IterCount_1 TO 1 BY -1  DO
  SWAP(BLK.Q_3[INTEGER(I)],BLK.Q_3[INTEGER(BLK.N_3+1-I)]);
Lab_10:   ;(* continue *)
  I := I + 1;
END;(* DO loop *)
END REVERS;  (* procedure body *)
END REVERS_0.
```

```
DEFINITION MODULE SWAP_0;
PROCEDURE SWAP(VAR A: REAL;VAR B: REAL);
END SWAP_0.

IMPLEMENTATION MODULE SWAP_0;
PROCEDURE SWAP(VAR A: REAL;VAR B: REAL);
VAR
  TEMP: REAL;
BEGIN  (* procedure body *)
TEMP := A;
A := B;
B := TEMP;
END SWAP;  (* procedure body *)
END SWAP_0.

DEFINITION MODULE Types;
TYPE
Logical = LONGWORD;
END Types.

IMPLEMENTATION MODULE Types;
BEGIN
END Types.
```

```
DEFINITION MODULE COMMON;
VAR
BLK:  RECORD
        CASE tag: SHORTCARD OF
        1:
           N_1: LONGINT;
           A_1: ARRAY INTEGER[1..200] OF REAL;
      |  2:
           N_2: LONGINT;
           A_2: ARRAY INTEGER[1..200] OF REAL;
      |  3:
           N_3: LONGINT;
           Q_3: ARRAY INTEGER[1..200] OF REAL;
      |  4:
           N_5: LONGINT;
           A_5: ARRAY INTEGER[1..200] OF REAL;
        END
      END;
END COMMON.

IMPLEMENTATION MODULE COMMON;
BEGIN (* module body *)
BLK.N_5 := 5;
BLK.A_5[1] :=REAL(5);
BLK.A_5[2] :=REAL(4);
BLK.A_5[3] :=REAL(3);
BLK.A_5[4] :=REAL(2);
BLK.A_5[5] :=REAL(1);
END COMMON.
```

7.2 Example 2

Fortran-77 source code:

```
      PROGRAM MAIN
*     initialize and find sum of an array
      DIMENSION VECTOR(10)
      INTEGER N
      DATA SUM/0/
      READ 30, N
```

```
  30  FORMAT (I2)
      CALL INITIALIZE(VECTOR,N)
      SUM = GETSUM(VECTOR,N)
      PRINT 20, SUM
  20  FORMAT (' The sum of the elements is ', F10.3)
      END

      SUBROUTINE INITIALIZE(V,N)
*     initialize 1st N elements of V
      INTEGER I
      REAL V(10), INC
      DATA INC/82/
      DO 99 I = 1,N
         V(I) = I + INC
 99   CONTINUE
      END

      REAL FUNCTION GETSUM(V,N)
*     returns sum of 1st N elements of V
      REAL V(10)
      GETSUM = 0
      DO 88 I = 1, N
        GETSUM = GETSUM + V(I)
 88   CONTINUE
      END
```

Generated Modula-2 code:

```
MODULE MAIN;(* main program module *)
FROM INITIALIZE_0 IMPORT INITIALIZE;
FROM GETSUM_0 IMPORT GETSUM;
FROM Types IMPORT Tarr_rl_10;
FROM FinOut IMPORT Format;
FROM FinOut IMPORT Read_Int;
FROM FinOut IMPORT Write_Rl;

VAR
  SUM: REAL;
  VECTOR: Tarr_rl_10;
  N: LONGINT;
BEGIN  (* module body *)
(* translation of DATA statements *)
```

```
SUM := REAL(0);
(* translation of executable statements *)
Format("I2");
Read_Int(N):
INITIALIZE(VECTOR,N);
SUM := GETSUM(VECTOR,N);
Format("' The sum of the elements is ', F10.3");
Write_Rl(SUM);
END MAIN.  (* main program module *)

DEFINITION MODULE Types;
TYPE
Tarr_rl_10= ARRAY INTEGER[1..10] OF REAL;
END Types.

IMPLEMENTATION MODULE Types;
BEGIN
END Types.

DEFINITION MODULE INITIALIZE_0;
FROM Types IMPORT Tarr_rl_10;
PROCEDURE INITIALIZE(VAR V: Tarr_rl_10;VAR N: LONGINT);
END INITIALIZE_0.

IMPLEMENTATION MODULE INITIALIZE_0;
VAR  (* "saved" variables *)
  INC: REAL;
PROCEDURE INITIALIZE(VAR V: Tarr_rl_10;VAR N: LONGINT);
LABEL Lab_99;
VAR
  I: LONGINT;
(* translator generated variables *)
  Start_1: LONGINT;
  Limit_1: LONGINT;
  IterCount_1: LONGINT;
  IterCtrl_1: LONGINT;
BEGIN  (* procedure body *)
(* DO I = ...  *)
Start_1 := 1:
Limit_1 := N;
I := Start_1;
IterCount_1 := Limit_1 - Start_1 + 1;
FOR IterCtrl_1 := IterCount_1 TO 1 BY -1  DO
```

```
  V[INTEGER(I)] := REAL(I)+INC;
Lab_99:   ;(* continue *)
  I := I + 1;
END;(* DO loop *)
END INITIALIZE;  (* procedure body *)
BEGIN (* module body *)
(* translation of DATA statements *)
INC := REAL(82);
END INITIALIZE_0.

DEFINITION MODULE GETSUM_0;
FROM Types IMPORT Tarr_rl_10;
PROCEDURE GETSUM (VAR V: Tarr_rl_10;VAR N: LONGINT): REAL;
END GETSUM_0.

IMPLEMENTATION MODULE GETSUM_0;
PROCEDURE GETSUM (VAR V: Tarr_rl_10;VAR N: LONGINT): REAL;
LABEL Lab_88;
VAR
  GETSUM: REAL;
  I: LONGINT;
(* translator generated variables *)
  Start_1: LONGINT;
  Limit_1: LONGINT;
  IterCount_1: LONGINT;
  IterCtrl_1: LONGINT;
BEGIN  (* procedure body *)
GETSUM := REAL(0);
(* DO I = ...  *)
Start_1 := 1;
Limit_1 := N;
I := Start_1;
IterCount_1 := Limit_1 - Start_1 + 1;
FOR IterCtrl_1 := IterCount_1 TO 1 BY -1  DO
  GETSUM := GETSUM+V[INTEGER(I)];
Lab_88:   ;(* continue *)
  I := I + 1;
END;(* DO loop *)
RETURN GETSUM;
END GETSUM;  (* procedure body *)
END GETSUM_0.

DEFINITION MODULE FinOut;
```

```
PROCEDURE Format(FormatString: ARRAY OF CHAR);
PROCEDURE Read_Int(VAR Val: LONGINT);
PROCEDURE Write_Rl(Val: REAL);
END FinOut.
```

Note that since the translator does not provide an implementation of the I/O procedures in `FinOut`, the implementation module of `FinOut` does not appear in the output from the translator in this example.

Chapter 8

Concluding Remarks

8.1 AGs and GAG

One drawback of AGs is that they are not modular. An AG cannot be partitioned into smaller pieces which can be worked on and tested separately. While a number of techniques for *composing* AGs have been proposed, the nature of a given AG remains *monolithic* ([Vog 89]). Information about every attribute occurrence in an AG is required to perform dependency analysis and determine an evaluation order, and all production rules are needed by the parser generator to generate parse tables. Consequently, in the development of an AG, a good deal of time is spent waiting for translators to be generated and compiled each time a modification to the AG was made (regardless of how minor the modification was).

With sizable AGs, tracking down circularities can be time-consuming and requires an overall familiarity with the AG. The source of a circularity introduced by even a subtle typing error can be hard to find as it can propagate through hundreds of productions in the AG.

Similarly monolithic is the Pascal program generated by GAG. Compilation was time consuming and strained the capacity of the Pascal compiler (which had to be extended in order to handle the generated translator).

ALADIN provided a sufficiently powerful range of types to represent all semantic information that was needed for translation. Moreover, it was generally easy to modify the AG to represent additional information by either extending the domains of attribute values or by introducing additional attributes into the AG. For instance, the domain of env was extended several times without great effort over the course of the translator's development.

Problems with the GAG system

While in general the GAG system proved to be powerful, a few troublesome features were encountered. These are summarized below.

- Generally the system is difficult to learn. While the manuals are quite thorough, the almost excessive formalism in the description (especially in the ALADIN manual) made learning to use the system more of a challenge than it need be. Examples are few and far between, and often features are described so formally that one has no idea how they might be helpful.

- In [Kas 82] we read that the generated programs are readable. Unhappily, this is truly only the case if German is a familiar language to the user[1]. To the unlucky user for whom it is not, identifiers such as `SYMLAENGENBEREICH` are far from mnemonic. This would not be especially important if it were not occasionally necessary (and in the author's experience, it occasionally *was*) to understand the roles played by such identifiers in the generated programs.

- Generally the diagnostics generated by the system are extensive and helpful, but error recovery is rather inelegant. Normally, if one of the passes detects unrecoverable errors the next pass is executed regardless, resulting in core dumps and leaving temporary files in the working directory which must be removed before GAG processing

[1]The GAG system has been developed at the Research Institute at the University of Karlsruhe, West Germany.

can be re-initiated. There are also cases when the system goes into an infinite loop (typically when some ALADIN type definition is illegal) before reaching the protocol phase, leaving the user with no indication of what caused the error. This was particularly unfortunate when the size of the input grammar was several thousand lines in length and GAG processing took upwards of thirty minutes, making debugging by trial and error a time-consuming exercise.

- While ALADIN typing rules may help cultivate a defensive programming style, they are in general annoyingly inflexible and awkward, especially when referring to discriminated union typed values. The lack of string handling operations is also inconvenient, rendering the use of `STRING` typed values next to useless. Since `STRING`s are stored in the translators' symbol table, they are no more like strings than symbols are, and there is little if any motivation for their use.

- PGS is not the best feature of the system. When a grammar input to PGS does not have the LALR(1) property, the user is left to guess where the problem is. PGS gives no indication of what kind of conflict it found in the grammar, nor does it indicate where it found a problem, nor does it make any attempt to resolve the conflict. Quite simply, it tells the user nothing at all and dumps core. In the development of the translator, it was necessary to maintain a version of the CF grammar in a form suitable for input to Yacc [Joh 78](which provides extensive information and error messages when conflicts arise).

  Perhaps the most serious problem encountered using PGS occurred when the number of symbols in the input grammar exceeded PGS's limit. While error messages indicated that the system could be extended by the modification of the values of some constants in the PGS source code, it was found that these features for extending the capacity of PGS simply did not work. Consequently it was necessary to write two versions of the translator, each implementing some but not all translation strategies.

8.2 Summary of restrictions on Fortran programs

The following list summarizes the restrictions which are placed on Fortran-77 programs by the translator.

1. Constant expressions may not contain exponentiation operations, character string operations, or complex arithmetic.

2. Integer constant expressions used (directly or indirectly) in array declarators and subscript expressions in EQUIVALENCE statements may contain only integer constants; these expressions are evaluated at translation time to enable the computation of offsets of associated entities from the start of the storage sequence with which they are associated. Alternatively, constant expressions could be generated for use in the declaration of the offset fields of records used to implement shared storage. For such an approach to be feasible, additional constraints would likely have to be imposed on the form of EQUIVALENCE statements (such as requiring the *first* entity in the list to share its first unit of storage with the first unit of storage in the entire storage sequence).

3. Arrays may be one-dimensional only. The translation strategies for both the representation of shared storage using variant records and the treatment of array arguments hinge on this constraint. While it does not seem feasible to uniformly preserve the dimensionality of Fortran arrays in the translation (since in Modula-2, actual and formal array parameters must have the same number of dimensions), the constraint could be removed by extending the translator to perform the conversion of multi-dimensional arrays into one-dimensional arrays. This conversion would involve declaring arrays which are multi-dimensional in Fortran as one-dimensional arrays with the same number of elements. The multiple subscript expressions which appear in array element references in the Fortran programs would subsequently have to be converted into single subscript expressions.

Perhaps the cleanest means of implementing this mapping of several subscript values onto one would be through the generation of functions, invocations of which replace multiple subscript expressions in an element reference. Each such function is associated with a particular number of dimensions (for instance, `map_2` would map all Fortran two-dimensional arrays onto one-dimensional arrays). Each of these mapping functions would take as argument a description of the dimension declarator used to declare a Fortran array, along with a number of subscript expressions. For example, a reference such as `A(INDEX1,INDEX2)` in the Fortran code, might be translated into `A[map_2(desc(A),INDEX1,INDEX2)]` in the Modula-2 code, where desc(A) represents information about the dimensions and subscript ranges of the corresponding Fortran array. Such mapping functions would essentially determine the represented element in an array element reference on the basis of the Fortran storage allocation scheme ([ANS 78], pg 5-6).

4. Logical and character entities must be explicitly typed (numeric entities need not). A more powerful scanner could eliminate the need for this constraint.

5. Numeric literals, identifiers, and Fortran keywords may not contain spaces. Again, a more powerful scanner could be employed to eliminate this constraint.

6. Character string expressions may contain only the concatenation operation; substring expressions are not handled but could easily be translated using Modula-2 library functions for string handling.

7. If a 'name-list' in a DATA statement contains an unsubscripted array name, then that name must be the only name in the list. The problematic case (see section 5.4.7) which this restriction prevents arises only when constant expressions are used in the declaration of an array whose initialization is specified using assignment statements in which subscripts are literals. The restriction could be removed by using only literals in the declaration of arrays which appear in DATA statements (as is done with arrays which appear in COMMON or EQUIVALENCE statements). Another

approach is simply to print a warning message when this situation is detected, informing the user that modification to certain constants may result in inconsistencies in the mboxModula-2 program.

8. If a repeat count is used in a constants' list of a DATA statement, then only one constant may appear in that list, and the repeat count should be equal to the number of locations specified to be initialized. This constraint, along with the restriction that implied DO-lists not appear in DATA statements, were introduced solely to keep the size of the translator reasonable. Both constraints could easily be eliminated with further elaboration on the current strategy employed in the translator.

9. The INTRINSIC statement is not handled, and consequently INTRINSIC functions may not be passed as argument. Since intrinsic functions are converted into regular functions in mboxModula-2, the passing of intrinsic functions as arguments could be handled in much the same way as external procedures are handled. Some additional care would, however, have to be taken in the generation of type identifiers for formal parameters corresponding to intrinsic functions since the types of the arguments used in their invocation may vary.

10. Invocations of intrinsic functions `MAX` and `MIN` cannot be translated because they take a variable number of arguments. However, if they were converted into a 'chain' of function calls each taking two arguments (for instance, the invocation `MAX(A,B,C)` can be converted into `MAX(MAX(A,B),C)`) then translation can be performed.

11. The ENTRY statement ([ANSI], pg., 15-12) is not translated. As each ENTRY statement effectively corresponds to a distinct subprogram, a possible strategy for its translation is to generate distinct subprograms for each ENTRY statement in a subprogram unit. This was not considered "reasonable" in this thesis.

12. RETURN statements which specify alternate returns may contain only integer literals. This restriction could be removed by converting RETURN statements with

integer expressions (which are not literals) into two statements, an invocation of a procedure which sets the appropriate boolean variable and a RETURN statement. The procedure would take the expression and all boolean variables corresponding to alternate return specifiers as arguments and set one of the boolean variables if the value of the expression is a positive number less than or equal to the number of alternate return specifiers in the subprogram.

13. Corresponding dummy and actual array arguments must have the same size and subscript range. This constraint can be eliminated by using open-array parameters for array arguments. Some modification of the subscript expressions in array element references would have to be performed in invoked subprograms to compensate for the zero-indexing of formal parameters.

14. Parts of arrays cannot be passed. That is, if an array element is supplied as actual argument, the corresponding dummy argument must be the same type as the array element. A possible approach to implementing the passing of 'partial arrays' is to pass additional information to subprograms which take "parts" of arrays as arguments. For instance, an offset value can be supplied as argument which is used within the subprogram to offset references to the corresponding dummy array elements.

15. Dummy procedures must be invoked at least once in the subprogram in which they are dummy arguments. As this restriction was introduced to enable construction of type identifiers for formal parameters which are procedures without global information, it can only be removed by a careful global analysis of procedure invocations. Such analysis involves tracing possible associations between dummy procedures and external procedures and consequently can be expensive.

16. Only the "short form" of input and output statements (ie., the PRINT statement and the READ statement without a control list) is translated. This restriction could be removed by extending the translator to handle the OPEN, CLOSE, and file positioning statements (as outlined in section 4.5.11) using the low-level file

handling procedures available on the Modula-2 system in the target environment. Subsequently the procedures for reading and writing in `FinOut` could be further parameterized to permit specification of file names (corresponding to device numbers in Fortran) and file positions (corresponding to the record numbers) for I/O with files connected for direct access.

17. Formatted internal data transfer (I/O from a buffer instead of an external device), is not handled. However, the same general strategy employed in the implementation of "format driven" I/O could be used to implement internal data transfers. Since the "buffers" used in internal data transfers are essentially character string variables, reading and writing to an internal buffer can be implemented by procedures which are similar to the reading and writing procedures in `FinOut` but which access substrings of character string variables instead of characters from an input stream; the process of converting these substrings into different representations according to format descriptions would be the same.

8.3 Results

The translator runs with speed comparable to a Fortran-77 compiler. In tests performed with fairly small Fortran programs (up to 500 lines in length) the time required to translate a given Fortran program into Modula-2 never exceeded the time required to compile the same program[2] by more than 33 per cent. These results suggest that there is room for expansion of the translator, and that an extended translator with fewer restrictions and acceptable performance can be created using the existing framework, provided a more powerful version of PGS becomes available. Note that one would likely tolerate a translator which is slower than a compiler since source-to-source translation is generally a *one-time* process used in the conversion of correct programs rather than in their development.

While it was not possible to compare the run times of Fortran programs and their

[2]using the ULTRIX Fortran-77 compiler on a Micro-VAX III.

Modula-2 counterparts (since the former were run on a mainframe and the latter on a PC), in terms of the number of statements executed Modula-2 programs are less efficient than the Fortran programs from which they were generated. The Modula-2 programs in general also have greater storage requirements as a result of the addition of auxiliary variables. A cursory examination of the examples in the previous chapter bears out this fact. Consequently, some degree of optimization of the Modula-2 programs seems to be in order. Such optimization can be performed both during translation (for instance, the translator could easily be modified to suppress generation of auxiliary variables for expressions in a DO statement when the expression is a literal) and as a post-translation step.

The AG formalism was found to be a powerful tool for specifying the translation and it appears that the description of source-to-source translation, at least between languages of the class of Fortran-77 and Modula-2, can quite effectively be achieved using AGs. Most difficulties which arose over the course of developing the translator involved incompatibilities of the two languages rather than with any shortcoming of AGs. The most troublesome (and underestimated) of these incompatibilities stems from the languages' treatment of arrays, and exemplifies a kind of problem which is bound to arise when translating from a language which is lower-level to one which is higher-level. The ability of the Fortran programmer to control the mapping of variables onto the memory of the machine is an example of a low-level detail which Modula-2 programs are not generally concerned with, as they deal with more purely abstract objects[3].

A nice feature of generated programs is their modularity. Since subsidiary modules are not buried in monolithic programs and are separately compilable, they can easily be *re-used*. Thus, part of the promise of the translator lies in the potential to translate large amounts of subprogram material available in Fortran libraries while preserving the 'library organization' of Fortran systems, thereby allowing more powerful Modula-2 systems to be

[3]Modula-2 was however designed for systems programming and provides features for low-level programming, although these features do not typify the high-level flavour of the language.

built.

References

[Alb 80] Source-to-Source Translation: Ada to Pascal and Pascal to Ada.
P. Albrecht, P. Garrison, S. Graham, R. Hyerle, P. Ip, B. Krieg-Bruckner.
SIGPLAN Notices, vol. 15, no. 12, 1980.

[ANS 78] *American National Standard Programming Language FORTRAN* (ANSI X3.9-1978).
American National Standards Institute, New York, New York, 1978.

[Boy 84] Lisp to Fortran - Program Transformation Applied. J. Boyle.
Program Transformation and Programming Environments.
NATO ASI Series, vol. F8, P. Pepper (ed.), Springer-Verlag,
Berlin-Heidelberg-New York-Paris-Tokyo,1984.

[DJL 88] Attribute Grammars - Definitions, Systems and Bibliography.
P. Deransart, M. Jourdan, B. Lorho. In *Lecture Notes in Computer Science*, no. 323, G. Goos, J. Hartmanis (ed.s), Springer-Verlag,
Berlin-Heidelberg-New York-Paris-Tokyo, 1988.

[Dob 87] SETL to ADA - tree transformations applied. S. Doberkat and U. Gutenbeil.
Information and Software Technology, vol. 29, no. 10, 1987.

[Eng 84] Attribute Evaluation Methods. J. Engelfriet.
Methods and Tools for Compiler Construction, B. Lorho (ed.),
Cambridge University Press, Great Britain, 1984.

[Far 89] A VDHL Compiler Based on Attribute Grammar Methodology.
R. Farrow, A. G. Stanculescu. *Sigplan Notices*, ACM Press, June, 1989.

[Fre 81] A Fortran to Pascal Translator. R. A. Freak.
Software - Practice and Experience, vol. 11, pp. 717-732, 1981.

[Gan 84] Attribute Coupled Grammars. H. Ganzinger and R. Giegerich. *SIGPLAN Notices*, vol. 19, no. 6, June 1984.

[Gle 84] *Modula-2 for Pascal Programmers.* Richard Gleaves. Springer-Verlag, New York-Berlin-Heidelberg-Tokyo, 1984.

[Goo 83] DIANA - An Intermediate Language for Ada. G. Goos, W. Wulf, A. Evans, K. Butler (ed.s). In *Lecture Notes in Computer Science*, vol. 161, Springer-Verlag, Berlin-Heidelberg-New York-Paris-Tokyo, 1983.

[Gro 86] *User Manual for the PGS-System.* J. Grosch, E. Klein. Research Institute at the University of Karlsruhe, 1986.

[Hut 87] *GCL : GAG Control Language.* B. Hutt, U. Kastens, E. Zimmermann. Research Institute at the University of Karlsruhe, 1987.

[Joh 78] *Yacc: Yet Another Compiler-Compiler.* S. C. Johnson. Bell Laboratories, Murray Hill, New Jersey, 1978.

[Kas 80] Ordered Attributed Grammars. U. Kastens. *Acta Informatika*, vol. 13, no. 3, pp. 229-256, Springer-Verlag, Berlin-Heidelberg-New York-Paris-Tokyo, 1980.

[Kas 82] GAG: A Practical Compiler Generator. U. Kastens, B. Hutt, E. Zimmermann. In *Lecture Notes in Computer Science*, vol. 141, Springer-Verlag, Berlin-Heidelberg-New York-Paris-Tokyo, 1982.

[Kas 87] *ALADIN - a Language for Attributed Grammars (version 7).* U. Kastens. Universitaet-GH Paderborn, Universitaet Karlsruhe, 1987.

[Ka2 87] *User manual for the GAG System (version 7).* U. Kastens. Universitaet-GH Paderborn, Universitaet Karlsruhe, 1987.

[Kel 84] Tree Transformation Techniques and Experiences. S. Keller, J. Perkins, T. Payton, S. Mardinly. *SIGPLAN Notices*, vol. 19, no. 6, June 1984.

[KHZ 87] *User Manual for the GAG System (version 7).* U. Kastens, B. Hutt, E. Zimmermann. Universitaet-GH Paderborn, Universitaet Karlsruhe, April 1987.

[Kin 88] *Modula-2: A Complete Guide.* K. N. King.
D. C. Heath and Company. Lexington, Massachusetts, Toronto, 1988.

[Ki2 88] *TopSpeed Modula-2 Language Tutorial.* K. N. King.
Jenson and Partners International, U.S.A., 1988.

[Knu 68] Semantics of context-free languages. D. E. Knuth.
Mathematical Systems Theory 2, pp. 127-145, 1968.

[Kri 84] Language Comparison and Source-to-Source Translation. B. Krieg-Bruckner.
Program Transformation and Programming Environments. P. Pepper (ed.),
NATO ASI Series, vol F8, Springer-Verlag, 1984.

[Leo 87] The Design and Implementation of a Converter Writing System.
S. Leong. *Proceedings of Miami Technicon 1987*, IEEE Miami, 1987

[Mei 82] *Fortran-77 Featuring Structured Programming.* L. P. Meissner,
E. I. Organick. Addison-Wesley Series in Computer Science.
Addison-Wesley Publishing Company, Massachusetts, California,
London, Amsterdam, Ontario, Sydney, April 1982.

[Mon84] How to Implement a System for Manipulation of Attributed Trees.
U. Moncke, B. Weisgerber, R. Wilhelm. *Fachtagung fur
Programmiersprachen und Programmentwicklung der GI*, Proc. 8,
Zurich, 1984. Informatik-Fachberichte 77, pp. 112-127, Springer, 1984.

[Pag 81] *Formal Specification of Programming Languages.* F. G. Pagan.
Prentice Hall Inc., New Jersey, 1981.

[Pet 73] On the capabilities of while, repeat, and exit statements.
W.W. Peterson, T. Kasami, and N. Tokura.
Communications of the ACM, vol. 16, no. 8, 1973.

[Sla 83] Conversion of Fortran to Ada using an Intermediate Tree Representation.
J. Slape and P. Wallis. *The Computer Journal*, vol. 26, no. 4, 1983.

[Tel 84] Production quality ADA compilers. J. Teller. *Methods and Tools for
Compiler Construction*, B. Lorho (ed.), Cambridge University Press, 1984.

[Ten 81] *Principles of Programming Languages.* R.D. Tennent.
Prentice/Hall International, London, 1981.

[TSD 88] *TopSpeed Modula-2 User's Manual.*
Jensen and Partners International, U.S.A., 1988.

[Vog 89] Higher Order Attribute Grammars. H.H. Vogt, S.D. Swierstra, M.F. Kuiper.
Sigplan Notices, ACM Press, June 1989.

[Wir 83] *Programming in Modula-2: Second, Corrected Edition.* Niklaus Wirth.
Texts and Monographs in Computer Science. David Gries (ed.)
Springer-Verlag, Berlin-Heidelberg-New York-Tokyo, 1983.

[Wir 85] *Programming in Modula-2: Third, Corrected Edition.* Niklaus Wirth.
Texts and Monographs in Computer Science. David Gries (ed.)
Springer-Verlag, Berlin-Heidelberg-New York-Tokyo, 1985.

[Yel 87] Attribute Grammar Inversion and Source-to-source Translation.
D. M. Yellin. In *Lecture Notes in Computer Science*, no. 302.
Springer-Verlag, Berlin-Heidelberg-New York-Tokyo, 1987.