THE ADA – COMPILER

On the design and implementation of an Ada compiler

J. van Katwijk

The Ada— compiler

On the design and implementation of an Ada language compiler

Proefschrift

ter verkrijging van de graad van
doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus,

prof. dr. J.M. Dirken,

in het openbaar te verdedigen ten overstaan van
een commissie aangewezen door het College van Dekanen
op dinsdag 15 september 1987
te 1400 uur
door

Jan van Katwijk

geboren te Amsterdam,
Wiskundig Ingenieur.

Dit proefschrift is goedgekeurd door de promotor

prof. dr. ir. W.L. van der Poel

De tekst van dit proefschrift is door de auteur ingetypt met de *ed* editor onder het UNIX† operating system. De tekst is geformatteerd met het volgende UNIX commando: pic | tbl | eqn | troff
en is afgedrukt op een Canon A1 laserprinter.

† UNIX is een handelsmerk van A.T. & T.

De Ada— compiler

Het ontwerp en de implementatie van een Ada† compiler

Samenvatting

In dit proefschrift worden enkele elementen uit het ontwerp en de implementatie van een compiler voor de programmeertaal Ada besproken. Daarbij wordt verwezen naar de *Ada— compiler*, een compiler voor bijna de gehele programmeertaal Ada. Deze implementatie is ontwikkeld aan de Technische Universiteit Delft, onder leiding van de auteur. De implementatie wordt *Ada—* genoemd omdat enkele details van de Ada programmeertaal (nog) niet zijn geimplementeerd.

In dit proefschrift worden de problemen bij de implementatie van een taal van de omvang en de complexiteit als de Ada taal besproken. De belangrijkste bijdragen van dit proefschrift zijn: (i) een overzicht van de literatuur over het implementeren van de Ada taal, (ii) een bespreking van algorithmes die toepasbaar zijn bij de implementatie van de Ada taal en die toegepast zijn bij de Ada— compiler.

Het proefschrift begint, na een inleiding, met een kort overzicht van de Ada taal. Daarna wordt een compiler model besproken dat voor een grote taal toepasbaar is en als model heeft gediend voor de Ada— compiler. De belangrijkste elementen van de Ada— compiler worden vervolgens in een grotere mate van detaillering besproken. De discussie over het front end van de compiler spitst zich toe op praktische aspecten bij *overload resolutie* en het efficient implementeren van de scope en zichtbaarheidsregels. De bespreking van het *back end* van de compiler valt in verschillende delen uiteen. Allereerst wordt het run-time model voor de beschrijving van data structuren besproken. Daarna wordt een overzicht gegeven van de vertaling van enkele van de Ada taal constructies. Deze beschrijving wordt gevolgd door een beschrijving van *low-level* talen voor de intermediaire programma representatie. Een apart hoofdstuk is gewijd aan het ontwerp en de implementatie van een *supervisor* voor de ondersteuning van Ada tasking in run-time. Ook is een apart hoofdstuk opgenomen waarin het ontwerp en de implementatie van een programma bibliotheek faciliteit wordt besproken. Voor de in de diverse hoofdstukken geformuleerde problemen worden oplossingen besproken.

† Ada is een geregistreerd handelsmerk van de regering der Verenigde Staten van Amerika, Ada Joint Program Office.

The Ada— compiler

On the design and implementation of an Ada language compiler

Abstract

In this thesis some elements in the design and implementation of a compiler for the Ada programming language are discussed. Reference is made to the *Ada— compiler*, an implementation of almost the whole Ada language. This implementation was made at Delft University of Technology under the author's supervision. It is called *Ada—* since a few details of the Ada language are not (yet) implemented.

In this thesis the problems are discussed that are encountered in the implementation of a language as large and complex as the Ada language. The main contributions of this thesis are: (i) a survey of literature on the subject of implementing the Ada language, (ii) a discussion of algorithms that are applicable in an Ada language compiler and that are applied in the Ada— compiler.

The thesis starts with an introduction and a brief survey of the Ada language. It then describes a compiler model suitable for the translation of a large language applied in the Ada— compiler. The major elements of the Ada— compiler are discussed in more detail. The discussion on the compiler front end is oriented towards practical aspects in overload resolution and the efficient implementation of the scope and visibility rules. The discussion on the compiler back end falls into several parts. First the run-time data description model is described. Then the translation of several higher-level language constructs is given. This description is followed by a brief description of low-level intermediate languages. A separate chapter is devoted to the design and implementation of an Ada tasking supervisor. Similarly, a separate chapter is devoted to the design and the implementation of a program-library manager. For the problems introduced in the various chapters solutions are discussed.

CONTENTS

1. Introduction

Writing a compiler is, even today, a rather complex job. Although since the early days of FORTRAN a tremendous progress has been made in the field, writing a compiler is still an art (or a craft) rather than a clerical job. Writing a compiler for a language of modest complexity, e.g. Pascal, is, using current methods and techniques, a relatively well-understood job. On the other hand, writing a compiler for a more complex language like the programming language Ada heavily relies on the inventivity of the compiler writer.

Some areas of the field are well understood; in these areas established formalisms can be used to describe or even to generate parts of a compiler. This is particularly true for scanning, parsing and to some extent for code generation. For the design of other parts, other formalisms can be used, either to develop a prototype or to increase the insight in how particular subprocesses are (to be) performed. In particular this applies to the use of attributed grammars or, to a lesser extent, to the use of denotational semantics. Nevertheless, in a recent paper [Boom-86] it is stated that current formal methods are not well-suited for use in programming language descriptions. We believe that it will take some time before such formal methods are ready to be used by a compiler writer.

For a large number of subtasks in the design and the implementation of a compiler, formalisms are not yet well-suited or even not applicable at all. This is by no means meant as a criticism of the use of formalisms; all that is stated here is that for the design and implementation of large-language compilers important parts have to be designed by the compiler writer himself.

The design and implementation of large-language compilers has some things in common with other software engineerings disciplines; a large part of the process is related to mastering the complexity of the problem and its solution. A positive side effect of designing large-language compilers is the increasing insight in how a compiler has to be built. It is this knowledge that can be used in a later stage as the basis for a process of automating parts of the compiler writer's job.

1.1 A review of the research

This paper makes reference to a project to design and implement a compiler for (a large subset of) the Ada language under the UNIX† operating system. As a result of the project, a compiler for almost the whole Ada language was designed, implemented and is operational. The implementation, called the *Ada— compiler*, is still incomplete (hence its name: the *Ada minus compiler*); it is expected, however, that in due course it will become a full Ada language compiler. The most significant omission in the implemented language is the lack of fixed and floating point arithmetic types and operators. Furthermore, the language differs in a number of minor details from the language described in [LRM-83]. A detailed survey of the implemented language is given in [Biegstraaten-87].

The research started as an exercise to implement a subset of the Ada language, more or less to satisfy our curiosity. The original subset, which was called DAS (*D*elft *A*da

† UNIX is a trademark of AT & T

Subset), included packages and overloading. It excluded tasking, generics, derived types, numeric types, stubs and separate units. After DAS was implemented successfully, the research aspect gained importance. It was not only felt interesting to implement a subset, it seemed at least as interesting to investigate the problems that would come up in the design and implementation of a larger subset or even of the full language. In particular, interest was raised for practical aspects concerning the construction of a compiler.

Currently, a number of validated Ada language compilers is available; the technology used in their construction is not widespread, however. This technology usually remains a company secret. One of the goals in this research is to make available algorithms, methods and techniques that are useful in the construction of Ada language compilers. The knowledge gained in the process of building a compiler can be used as a basis for improving the implementation itself. Furthermore it should be the basis for further research in and development of compiler technology.

1.2 A review of this thesis

No attempt is made in this thesis to write a *Cookbook for the Ada-compiler builder.* Our implementation is currently about 55,000 lines of C code and although only a small part of the code is really interesting, this small part is still over 10,000 lines of code. Such an amount is far too large to allow a detailed discussion. Therefore, the description is restricted to a review of some particularly interesting points. As such, the choice resulted in a discussion on compiler front-end aspects, a discussion on aspects of the compiler back end, a discussion on the design and implementation of a tasking supervisor and a discussion on support for separate compilation.

— Front end: The front end of an average Ada language compiler deals with various forms of analysis that are required by the language reference manual [LRM-83] and generates a high-level intermediate program representation. Points that were thought to be particularly interesting are:

  - practical aspects of overload resolution. Although the basic approach to overload resolution in Ada is well known, any implementor has to solve a number of practical problems. Problems that can be identified are: resolving the syntactic ambiguity in indexing, calling and slicing operations and handling of special operators such as e.g. the equality operator, which is implicitly declared for user-defined types in the Ada language.

  - the implementation of scope and visibility rules. The Ada language has forms of explicit scope control†. Managing a compile-time symbol table is complex and time-consuming. A straightforwardly implemented set of algorithms for symbol table management consumed about 25% of the time spent in front end processing.

— Back end: The back end of an Ada language compiler performs a translation of the high-level intermediate program representation obtained by the front end to target assembler code. A major effort in our work was the design and subsequently the

† Explicit scope control gives the programmer control over the visibility of declarations in terms of the program text.

implementation of a model for the representation of the complex data structures at run time. This model, the *doublet model*, turns out to be simple and easy to implement while yielding fairly efficient run-time code.

— Tasking: Tasking is considered to be an important as well as a complex topic.

— Separate compilation: One of the contributions of the Ada language to the field of software engineering is the safe separate compilation facility. This facility requires support in the form of a program library.

The organisation of this thesis is therefore as follows:

- first a short chapter (chapter 2) is devoted to a discussion of some of the main characteristics of the Ada language;
- in the next chapter (chapter 3), an overview is given on the architecture of a compiler for the Ada language together with a brief overview on literature on the topic.
- the bulk of this thesis is formed by four technical chapters (chapters 4, 5, 6, and 7), describing some details of algorithms that are applied in the implementation of the Ada— compiler. Chapter 4 is devoted to elements of the front end; chapter 5 to elements of the compiler's back end, chapter 6 to various (implementation) aspects of tasking in the Ada language and chapter 7 to separate compilation and its implementation;
- finally, a short chapter (chapter 8) is dedicated to results and conclusions.

1.3 Motivation for the present work and a project history

There are, of course, several reasons for implementing a language as large and complex as the Ada language.

Prior to attacking the Ada language, the author had implemented Algol 60 twice. The first implementation was in BCPL [Richards-80] on a DEC/PDP-11 computer under the RT/11† operating system. The second implementation, a redesign of the previous one, was written in C under the UNIX time-sharing system. Having implemented ALGOL 60 and having interest in systems programming and programming language implementation, it was felt that the design and implementation of a compiler for a (small) subset of the Ada language could be a real challenge.

Interest for the Ada language was expressed in early 1980 when the author organized a workshop on the language. As a result, other staff members became highly interested, as did students; the interest centered on the aspects of software engineering, raised by Ada as language, by the language itself and by its implementation aspects.

Some students began to design and to build a simple context-free parser for Ada. At that time discussions started among staff members on the feasibility of implementing an Ada subset as a student project. A group of four students (Henri Bal, Wim de Pauw, Hans van Someren en Jeanet Vonk) was asked to investigate the possibilities for such an

† PDP, VAX and RT/11 are trademarks of Digital Equipment Corporation

implementation. Later, three of them (Henri Bal, Hans van Someren en Jeanet Vonk) actually began with the design and implementation of a subset which was called DAS.

Henri Bal and Jeanet Vonk designed a front end [Bal-82], [Vonk-82]. The resulting front end which was written in C ran on a PDP-11/60 under the UNIX time sharing system. It consisted of two separate programs:

- a lexical and syntactical analyzer comprising the first pass, [Vonk-82];
- a static-semantics analyzer comprising the second pass [Vonk-82], [Bal-82].

At the same time a tree-structured intermediate program representation based on early versions of DIANA† [DIANA-83] was designed. The output of the first compiler pass was a program representation in this intermediate notation; the second pass operated on the intermediate program representation, essentially without changing its structure.

One of the first steps in the design of the back end of our compiler was the development of a model for the description of data at run time. We felt that the classical dope-vector based approach as described by e.g. Gries [Gries-71] was less suited for a language like the Ada language. Dope vectors carry a pointer to the object they describe or are part of this object. Sharing descriptors between objects with a similar structure and similar constraints seems hardly possible. Furthermore, we felt that the usual dope-vector based tree-structured organization of complex data structures accompanying the dope vector approach would complicate the generating of efficient access code to components of these data structures. A new model, the doublet model, was designed by this author in cooperation with Hans van Someren [Katwijk-84a].

A start of the implementation of the computation of storage requirements based on the doublet model was made by Hans van Someren. He implemented a part of the *storage allocator*, a compiler phase determining for each run-time object its storage class and its addressing path [Someren-82]. Wim de Pauw made a first attempt to implement an *expander phase*, a compiler phase mapping the high-level intermediate program representation onto the low-level intermediate program representation of the selected code generator [Pauw-83].

An interpreter for (a part of) DAS was designed and implemented by Niels Bogstad en Albert Hartveld [Bogstad-83]. The work was carried out at the Department of Electrical Engineering under the supervision of Prof. A. van de Goor and Hans van Someren. The interpreter was based on a literal implementation of the doublet model. As could be expected, it was extremely slow. Nevertheless, at that time it was helpful to have such a tool available.

A major restriction in the design and the implementation of the compiler was the 64 Kb address space of the PDP-11/60. There was no possibility to have any significant part of the intermediate program representation in main memory. As a result, the intermediate program representation was constructed piece by piece and was processed piece by piece.

† DIANA, *D*escriptive *I*ntermediate *A*ttributed *N*otation for *A*da, was intended as a standardized intermediate program representation.

The work thus far was done by students as part of an assignment for their master's thesis project. In spite of the complexity, these assignments were performed quite well. However, soon the (code) size of the static-semantics analyzer was too large to fit the memory of the host computer and a decision had to be made as whether to stop the project or to redesign the compiler. Since it was felt that a reasonable subset of the Ada language could be implemented, it was decided to redesign and to rewrite the front end. Redesign and re-implementation were done by the author. The resulting front end was functionally improved. It consisted of three programs:

- a lexical and syntactical analyzer;
- a first part of the static-semantics analyzer consisting of a *reference resolver* handling declaration processing and identifier look-up;
- a second part of the static-semantics analyzer consisting of a type checker handling the overload resolution.

In the same period Wim de Pauw continued the work of Hans van Someren and completed the implementation of the computation of storage requirements. He made a start with the generation of low-level intermediate code [Pauw-83]. Prior to this stage of the development of the compiler it was decided that an existing code generator should be used. This decision was taken since it was expected that reuse of an existing code generator would save considerable time and effort. With regard to the operating environment, usage of the code generator of the portable C compiler seemed a natural choice.

Chuck Barkey continued the work of Wim de Pauw. He built a significant part of the expander [Barkey-83]. Later on, he contributed to the implementation while working for TNO-IBBC.

A first implementation of a program-library manager was made by Maarten de Niet. His implementation was based on a preliminary investigation made by Hans van Someren [Someren-82b]. This implementation was finished and made operational in the summer of 1983 by the author. The implementation was extended and improved by Robert van Liere.

During the summer of 1983 the first DAS programs actually ran on the PDP-11/60. Not including the code generator (borrowed from the PCC, the Portable C Compiler) the compiler had five passes. Schematically:

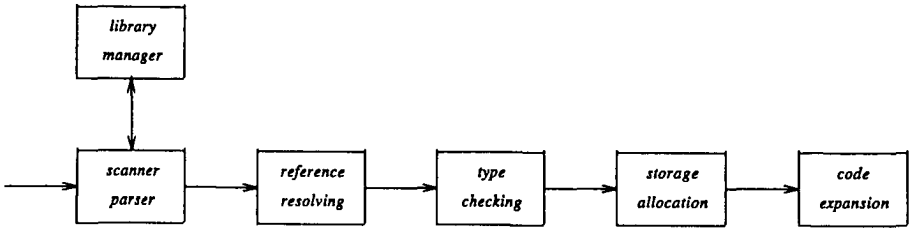


Figure 1. *The five passes of the DAS compiler*

The (large) number of passes was completely determined by the limited amount of main memory. Separation of functionality was enforced by memory limits rather than being the result of a particular design strategy.

Since it was nevertheless apparent that only a small subset of Ada could be handled on a machine of such a limited size, an M68000† -based computer system was acquired and the implementation was ported. As part of the master-thesis assignment of Maarten de Niet [Niet-84], the three programs comprising the front end were merged and their functionality was improved. The resulting front end was syntax driven; on the fly, an intermediate program representation was generated (in memory) and checked for consistency. YACC- [Johnson-74] generated parsers were extended with error-repair capabilities by Hans Toetenel [Toet-84a]. In the meantime Klaas Dijkstra implemented a tracer for DAS programs [Dijkstra-84].

As soon as a transition was made to an M68000-based machine, the need for another code generator arose. On a UNIX distribution tape sources for a version of *a* portable M68000-based C compiler (the *MIT* back end) were found. These sources turned out to be quite useful; with little effort a code generator for an M68000-based instruction set could be constructed and adapted to the requirements imposed by the expander.

At the end of 1983 improved algorithms for the generation of code in the doublet model were designed and implemented [Katwijk-87a]. Hans Toetenel extended and modified the doublet model to support default initialisations of record components, access values and allocators; he further designed a first implementation of aggregates [Toet-84b]. Since then the compiler back end has undergone two major revisions. The first one was performed by the author; algorithms for aggregates and access values were redesigned and reimplemented. The second revision, a major one, was carried out by Hans Toetenel. The purpose of this latter revision was to interface the compiler back end to a new version of the code generator of the portable C compiler.

† M68000 is a trademark of Motorola Inc.

The MIT code generator had been in use for over a year before a transition was made to the code generator of the regular C compiler on our system. The latter C compiler was a slightly different version for which no sources were available. A major reason for this transition was to retain compatibility with the output of other compilers on the system for (almost) free. In particular the ability to call C functions was found important.

As an experiment in retargetability an attempt was made to retarget the compiler to EM code and to use existing EM code generators to generate actual target code (EM is a family of intermediate stack-based languages, originally designed for Pascal by Tanenbaum et al [Tanenbaum-83a].) A first attempt was made by Erik van Konijnenburg [Konijn-84]. He used a modified version of the portable C compiler-code generator to generate EM code, leaving the expander almost unmodified. The generated EM code was translated to M68000 assembly code by an available PCC-to-EM code generator. For a variety of reasons this attempt was hardly successful. In a second attempt, by Wim Moermans [Moermans-85], the expander was modified to generate EM code directly. This attempt was reasonably successful. Based on this experience, an experimental version of the compiler back end using EM as an intermediary was built [Barkey-85] although this back end was never completed. In spite of the fact that the original design of the expander was oriented towards the prefix-encoded tree representation used by the portable C code generator, less than 10 % of the expander had to be rewritten for the generation of EM code. Recently Ronald Huijsman, Douwe Kamstra and Kees Pronk took up this lead to create a complete back end of the compiler using EM as intermediate code.

Since the work of Maarten de Niet the compiler front end has undergone several major revisions. The revisions improved the functionality (e.g. addition of integer types, of derived types, of tasks, of generics). Furthermore, a variety of algorithms and techniques was experimented with (e.g. symbol table management, overload resolution).

Tasking was included in cooperation with Hans Toetenel. A first design and implementation of a tasking run-time supervisor was given by the author in close cooperation with Hans Toetenel. (Hans Toetenel made the tasking supervisor operational.) The suggestion to describe the allowable sequences of supervisor calls by means of syntax charts was made by Hans van Someren. Further research into efficiency aspects of tasking supervisors for the Ada language is currently being carried out by Ronald Huijsman, Hans Toetenel and the author [Huijsman-87].

The program-library manager is required for separate compilation. An early version was implemented by Maarten de Niet and the author. Since then some experiments with implementation models were performed by Robert van Liere when working for TNO-IBBC [Liere-87]. Recently, functional improvements and extensions were made by Ton Biegstraaten.

Klaas Dijkstra's DAS tracer was adapted and extended by Ton Biegstraaten.

The current Ada— compiler implements almost the whole Ada language. It is still written in C and consists of about 850 lines of syntactical specification and 55,000 lines of C code.

Making an implementation for the Ada language is a major undertaking; an attempt to implement the language was felt to be a challenge in many respects. A number of reasons can be identified why this undertaking was started:

- First of all, there was an interest in the problem of tackling the implementation of various language constructions. In particular from the engineer's viewpoint: what price must be paid for the compilation and execution of the various language constructions? Would it be possible to define and implement a flexible portable run-time system for the support of the complex data structures that the Ada language allows? Is it possible to achieve a Pascal-like run-time efficiency for constructions that are limited to a Pascal complexity at a fair price in compilation-time?
- Second, there was an interest in the problem of mastering a project of such a complexity. Could a truly usable implementation of such a large-language (though perhaps in a prototype form) be achieved in our academic environment?
- Third, there is a strong interest in the whole area of compiler technology. The emphasis of our interest is on practical compiler technology. After all, our environment is a *university of technology*. One of the advantages of building a large-language compiler is that areas where formal methods are applicable can be as clearly identified as the areas where other kinds of technology should be used. Given the fact that the Ada language is large, probably the last of the large imperative languages, it seems an excellent vehicle for exercising practical compiler technology.

The design and the implementation of our compiler has not been an isolated activity. Compiler design, however important it may be, should always be related to the study of programming languages and the study of the environments in which the compiler is to be developed and in which it is going to be used.

1.4 Research goals

It has almost been a tradition to have the sources of a number of Pascal translators more or less in the public domain. The popularity of Pascal is mainly due to the wide-spread availability of complete compilers. Even now, several textbooks on compilers discuss complete Pascal translators. Recently a complete description of the source of a model implementation for Standard Pascal (IS-7185) has been made available in the form of a book [Welsh-86].

For the Ada language the situation is completely different. Most compiler developers or vendors cannot be persuaded to disclose any real technical data on their compiler. (Exceptions are: (i) the NYU-Ada translator - under a licensing agreement the sources of the NYU-Ada compiler/interpreter are available - and (ii) the Karlsruhe implementation. Uhl et al [Uhl-82] present a complete attributed grammar for the Ada language from which a semantic analyzer was derived.) A large number of companies developing compilers for the Ada language was invited to send some technical data on their product. Most of the companies addressed did not even bother to answer the request. From time to time, however, pieces of technical data were disclosed in private conversations with people from such companies.

The attitude of these companies is not really surprising. The implementation of a production-quality Ada language compiler requires large investments and disclosure of implementation details may have advantages for the competition. A consequence of this attitude is that knowledge on the technology used for Ada implementations is far less widespread than the knowledge on technology for e.g. Pascal implementations. This shows i.a. in the amount of available literature. The number of descriptions of techniques applied in the construction of Ada language implementations is relatively small. Of these papers, most of the earlier ones are virtually impossible to obtain. A number of U.S. institutions was requested to send some older material, the result was quite disappointing.

The objectives of this research were twofold. First, in surveying the literature related to the implementation of the Ada language; second, in developing technology and algorithms in the form of a prototype Ada language compiler. Some restrictive remarks are in order. Although it is held that a survey of *most* of the literature on Ada implementations is given, it is almost certain that the survey is still incomplete. It is nevertheless thought that the surveyed literature presents the current state of the art in the design and implementation of Ada language compilers.

The second main contribution of this thesis is the discussion of several of the more complex aspects in the design and implementation of an Ada language compiler. We present a survey of problems *and feasible solutions* for the problems with overload resolution, efficient identifier lookup for languages with explicit scope control, declaration and instantiation of generics, code expansion for complex data structures, tasking implementation and the implementation of separate compilation.

2. A short overview of the Ada language

The history of the Ada language is well known. Within the United States Department of Defense (DoD), a large number of programming dialects was used on a tremendous amount of different computer systems. In 1973 a group of people began to investigate the possibility of standardizing on a single programming language for all DoD embedded computer applications. After several iterations over the requirements for such a language (Strawman, Woodman, Tinman, Ironman and Steelman), contracts were awarded for the development of four languages. These languages were given names of colours: the *red* language (designed by Intermetrics), the *green* language (designed by Honeywell/CII), the *yellow* language (designed by SRI) and the *blue* language (designed by SofTech). In 1979 the winning language was selected and Brian Wichmann, one of the members of the *green* language design team, could write *Ada is green* [Wichmann-79]. The winning language was named after Ada Augusta, Lady Lovelace, daughter of Lord Byron. Ada Augusta (1815 - 1852) was a friend and assistant of C. Babbage, the inventor of the *analytical engine.* Ada Augusta is generally considered to have been one of the very first programmers.

The *green* language was published as *preliminary Ada* in a special issue of Sigplan Notices [Ada-79], [Rationale-79]. The first major revision of the language appeared in 1980 [Ada-80]; in early 1983 an ANSI (American National Standards Institute) standard appeared [LRM-83]; since then, work is being done to make this ANSI standard into an ISO (International Organisation for Standardisation) standard.

At later stages in the language design, attention was paid to the capabilities of a program environment in which Ada language programs could be designed, implemented, and maintained. Requirements for such APSEs (*Ada Programming Support Environments*) were given in the *Stoneman* report [Stoneman-80].

It is impossible to describe a language of the size and the complexity of the Ada language in just a few pages. Nevertheless, we shall try to give the reader who is not familiar with the Ada language a short overview of some of the main characteristics of the language.

Origin

To understand the structure of the Ada language, it is important to know that this language has its roots in Pascal. This is not accidental; it was one of the requirements for the language. The relationship with Pascal causes the language to be a statement language and also to have a global structure which is similar to that of Pascal.

The remainder of this chapter deals with a discussion of the following topics:

- overloading;
- scope and visibility rules;
- generics;
- complex data structures;

- exceptions;
- tasking;
- separate compilation.

Overloading

Overloading is a language feature that allows a symbol in the program to be ambiguous when only looking at the scope and visibility rules. Overloading as such is far from recent; even ALGOL 60 had its numerical operators overloaded. The exact meaning of an applied occurrence of e.g. a "+" operator symbol in an ALGOL 60 program could not be determined without taking the types of the operands into account. In the Ada language user-defined function identifiers and enumeration literals may be overloaded: several functions and enumeration literals with the same identifier may be visible at the same time at a given point in the program.

Overload resolution, i.e. the determining which declared entity is meant by the use of a certain identifier in an applied context, is done by taking the context into account. Consider the following example in which function identifiers and enumeration literals overload each other.

```
declare
  type colour is (red, white, blue);
  type paint is (red, blue, yellow, white);
  a : colour;
  function red (x : colour) return colour is
  begin
     null; -- some statements
  end;
begin
    a := red (red);
    ...
```

Three interpretations could, it seems at first glance, be given to each of the occurrences of *red* in the statement

```
a := red (red);
```

Taking the context into account leads to unique interpretations for both occurrences: the first occurrence of *red* is identified by the function identifier, the second one by the *red* literal of type *colour*.

Scope and visibility rules

The scope and visibility rules define the scope of declarations and the rules according to which identifiers are made visible in various points in the program.

For each form of declaration the language rules define a certain portion of the program text to be the *scope* of the declaration. Declarations may occur in e.g. package

specifications; the scope of these declarations extends to the corresponding body construct. Consider this example of a package:

```
package p is
  ...
  a : integer;      -- immediate scope of this declaration
                    -- extends to the end of the specification
  ...
end;
```

The scope of *a* extends towards the corresponding body, regardless of the place where the latter appears.

The scope of a declaration that occurs immediately within a declarative region extends from the beginning of the declaration to the end of the declarative region; this part of the scope of a declaration is called the *immediate scope.* The scope of a declaration may extend beyond the immediate scope, e.g. in a package the scope of a declaration in the visible part extends to the scope of the enclosing declaration.

A declaration that is elaborated within the visible part of a package is *not* directly visible outside this package. To access such a declaration, it must be explicitly *selected.* An alternative is to make all declarations in the visible part of a package visible by mentioning the package name in a *use clause.*

Accessing a declaration through selection takes the form of a record selection, e.g.:

```
p. a;
```

defines the *a* in *p* in any context where p is visible. Visibility through selection can also be used *inside* any form of labelled constructions; only declarations in the visible part of packages can be made visible outside the construction. (Notice that neither declarations in the private part of a package specification nor declarations local to the corresponding package body can be made visible outside the construction in which they appear.)

A *use clause* achieves direct visibility of declarations that appear in the visible part of named packages. Consider

```
use p;
```

After the elaboration of this use clause, the declaration of *a* in the package *p* can be accessed by writing the identifier *a.*

Declarations appearing in open scopes can never be hidden by declarations made directly visible through a *use clause.* Consider again

```
use p;
```

In an environment

```
declare
  a : integer;
  use p;
begin
  ...
```

the *a* defined in package *p* remains hidden by the *a* declared in the block.

A second constraint in the use of *use* clauses is that potentially visible declarations in different packages may hide each other. Consider a second package *q*,

```
package q is
  ...
  a : integer;
end;
```

The mentioning of both *p* and *q* in a use clause

```
use p, q;
```

causes both the *a* from *p* and the *a* from *q* to hide each other.

Generics

Generic units provide a means to parameterize pieces of program on values, types, and operations. Consider the example of a generic queue package:

```
generic
  type queue_element is private;
package queue is
  type queue is private;
  function new_queue return queue;
  -- other queue operations
end queue;
```

The declaration of a generic unit defines a *template* rather than a real package. Prior to using a queue package, as in the example above, an *instantiation* has to take place. In an instantiation the actual generic parameters are provided and (conceptually) a package is created. Consider the instantiation

```
package int_queue is new queue (integer);
```

The result of the elaboration of this instantiation is a package, named *int_queue*, defining a data type *queue* (with *integer* queue elements) and some operations on values and objects of this type. Since virtually *any* type can be used in the instantiation, a subsequent

instantiation could be:

```
package int_queue_queue is new queue (int_queue. queue);
```

Result of this instantiation is a package *int_queue_queue* with as *queue_element* type the type *queue* from the previous instantiation.

There is a number of restrictions on the definition and the use of generics.

Complex data structures

The Ada language supports arrays and record structures. Unlike Pascal, type definitions may be parameterized (constraints). In particular array types and record types may have to be constrained (i.e. provided with parameter values) before they can be used to declare objects of the type. Consider the following example in which an array type and a record type definition are given:

```
subtype int is integer;
type aint is array (int range <>) of int;

type u_rec (a : int) is record
  x : aint (1 .. a);
end record;
```

The parameters, the *discriminants* of the record type, can be used to constrain subtypes appearing as component types or they can be used to select in a variant record.

Exceptions

The Ada language supports exceptions "*for dealing with errors or other exceptional situations that arise during program execution*". To *raise* an exception is to abandon normal program execution, so as to draw attention to the fact that the corresponding situation has arisen.

An exception is handled by an *exception handler.* An exception handler is a user-provided piece of program attached to a block, a package body, a procedure body or to a task body. An exception itself is user-declared, although some exceptions are predefined. Consider the following program fragment in which an exception is declared and an exception handler is specified:

```
declare
  some_error : exception;
begin
  --sequence of statements
exception
  when some_error => -- sequence of statements
end;
```

An exception raised within the execution of the sequence of statements of a block causes

control to be transferred to the exception handler attached to the block. If the exception is e.g. *some_error*, it is handled by the exception choice labelled *some_error*, otherwise the execution of the block is terminated and the exception is passed to the enclosing block.

Tasking

The Ada language allows certain program units, tasks, to operate in parallel and to communicate with each other during their execution. Tasks may appear as local objects or as access objects.

Tasks are defined as types; their form is similar to the form of packages. A definition of a task type falls into two parts. The first part, the task-type specification, specifies the name of a task type together and gives the specifications for the entries of the task. The second part specifies the code to be executed, i.e. the task body. Consider the example of a task-type specification:

```
task type queue_task is
  entry add_elem (v : queue_elem);
  ...
end queue_task;
```

In this task-type specification it is specified that task objects of the type own an entry *add_elem*. Declaration of an object *queue* of type *queue_task* causes a task to be created, (eventually) to be activated and to run as an autonomous process:

```
queue: queue_task;
```

Task communication is supported by a *rendezvous mechanism*. The caller calls upon an entry of a particular task object; the code belonging to this entry, the *accept body*, belongs to the called task. Data transfer between the caller and the callee can take place through ordinary parameter transfer. In a rendezvous two tasks share the execution of the accept body.

A call to the entry *add_elem* of the task object *queue* takes the form

```
queue. add_elem (actual parameter expression);
```

During the execution of the rendezvous the calling task is suspended, i.e. an entry call is a *synchronous* call.

A task body for *queue_task* could look (schematically):

```
task body queue_task is
  ...
  accept add_elem (v : queue_elem) is
  begin
    ....
  end;
```

```
end;
```

Separate compilation

The Ada language supports the notion of a safe separation compilation to support programming in the large. This separate compilation facility allows type checking to be done over separately compiled modules. Two forms of separate compilation can be distinguished: first, separately compiled packages and subprogram specifications and their bodies and second, *sub*-units.

Separately compiled packages and subprogram specifications are kept in a *program library*. Data on compiled compilation units in a program library can be made available to other compilation units by mentioning the name of the compilation unit in a *with clause*. Consider the example of a package *p* acting as a compilation unit:

```
with text_io;
package p is
  ...
end;
```

The *with clause* specifies that on compilation of this compilation unit declarations in the compilation unit *text_io* must be made available.

The program library manager is responsible for maintaining the consistency of the program library contents. Changing a library unit (e.g. a package) and recompiling it successfully must cause the invalidation of all compilation units that depend on this library unit. (Informally stated: a unit depends on another unit if the former needs something from the latter for its compilation.)

Sub-units are bodies that are to be included in another body. Consider the example of a package body *p* with a *stub*:

```
package body p is
  ...
  procedure q is separate;
  ...
begin
  q;
end q;
```

In this body the specification for the procedure *q* appears as a stub. Its specification makes the procedure visible. The body of *q* can be compiled later. Sub-units support a top-down design paradigm.

3. Architecture of an Ada language compiler

3.1 Introduction

In this chapter the architecture of the Ada— compiler is discussed. First a brief review is given on the architecture of compilers in general; second, the attention is focused on Ada language translators and the Ada— compiler. The chapter is concluded with a review on the literature of existing compilers for the Ada language and subsets of the language.

For an overview of the terminology used, the reader is referred to the literature on compiler technology, e.g. [Aho-85].

Any compiler can be split up into two major phases:

- an analysis phase;
- a synthesis phase.

The analysis phase reads the source program and checks whether the conditions imposed by the syntactical and semantical definitions are fulfilled or not. The output of the analysis phase is conceptually some annotated parse tree, the *Abstract Syntax Tree* (AST).

The synthesis phase translates the AST into a program in some target language. This target language may be assembly code, machine code or some other higher level language.

A more detailed classification of compiler phases is given in figure 2. The analysis phase of a compiler encompasses two major functions: *lexical and syntactical analysis* and *static-semantics analysis.*

The *lexical analyzer* reads the source program and identifies the *tokens* or the *symbols* it contains. The input to the syntactical analyzer is a stream of tokens rather than a stream of characters. Lexical analysis is often based on regular expressions or regular grammars [Aho-85].

The *syntactical analyzer* reads the token stream and groups the symbols into grammatical phrases that are used by the compiler to synthesize output. Usually, the grammatical phrases of the source program are represented by a *parse* tree. Syntactical analysis is often based on context-free grammars [Aho-85].

The *static-semantics* analyzer checks the source program for consistency with the static-semantics of the programming language. The term *static-semantics* is a misnomer, it has hardly anything to do with semantics of the language. The term is heavily used in the literature to indicate the *context conditions* and the static properties of programs in the language [Aho-85]. Throughout this thesis we use the term with that meaning. The static-semantics analyzer gathers type information for type checking of the expressions and their constituents. The output of the static-semantics analysis phase is (at least conceptually) an annotated parse tree. In this annotated parse tree the implicit dependencies that exist in the source program are made explicit. The semantic analyzer is usually hand-crafted; it is seldom based on formal methods or techniques. Research on the practical application of formal methods, e.g. denotational semantics and attributed grammars, is progressing, though. In particular attributed grammars can be used to

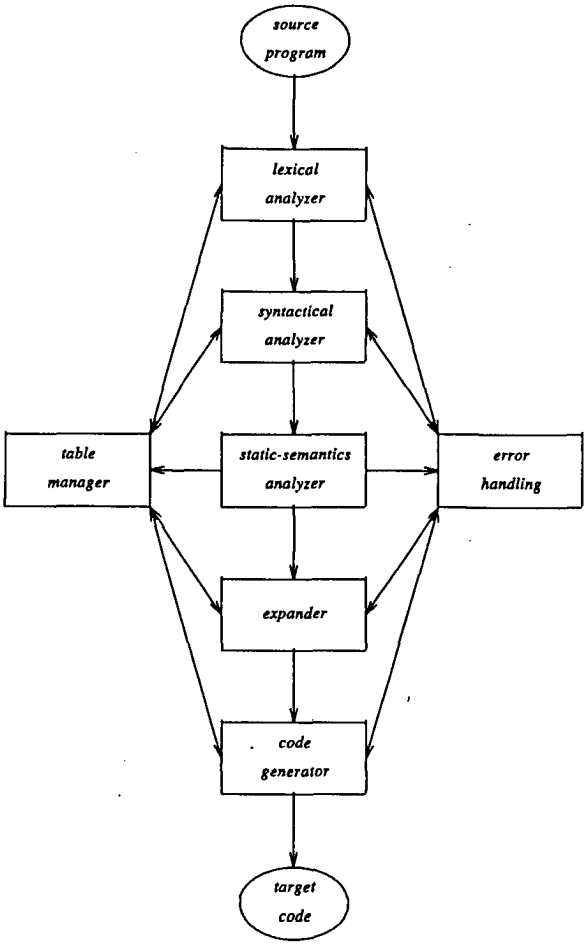


Figure 2. *Structuring of a compiler*

prototype static-semantics analyzers.

For small-language compilers, e.g. compilers for Pascal or Pascal-like languages, the static-semantics phase drives the code generator directly; it does not produce an explicit high-level intermediate program representation. On the other hand, large-language compilers often create an explicit intermediate program representation stored on an external medium.

The synthesis phase of a compiler encompasses the construction of a program, semantically equivalent to the input program, written in some other language, the *target* language. Often this translation is performed in two or more phases (as indicated in figure 2): the expander phase and the code generator.

The *expander* reads the intermediate program representation which was constructed by the static-semantics analyzer and generates an equivalent program on a lower semantic level. Operators and operands that were represented in terms of source level entities in the input are translated into constructs and expressions encoded on the level of the target code. As an example, a variable which is represented in the input by a reference to the defining occurrence of that variable, is translated into access code to the run-time location of that variable.

Finally, the *code generator* generates target machine code or target assembly code.

Between the expander and the code generator an optimizer may be applied. Similarly, a peep-hole optimizer may be applied to the code generator output.

The analysis phases of a compiler check conditions on which they may encounters errors in the user program. In the model depicted in figure 2, error handling is centralized in a separate module.

During the compilation, information gathered in the analysis will be used in other parts of the compiler. This information is stored in tables.

Further aspects to take into account by a compiler are the support for separate compilation and run-time support, e.g. tasking support.

3.2 The architecture of the Ada— compiler

In most known compilers for the Ada language the analysis phase and the synthesis phase are implemented as separate programs (or even as separate sets of programs). Analysis is performed by the compiler *front end*, synthesis by the compiler *back end*. The compiler front end transforms a source program into a program in a high-level intermediate representation (e.g. DIANA). The compiler back end takes the program in this intermediate representation and translates it into the target code.

Although it is possible to combine the front end and the back end in a compiler for a language of the complexity of the Ada language, this is seldom done. Splitting the compiler into separate programs for front end and back end has a number of engineering advantages. The static-semantics of the Ada language are complex; a front end that only deals with the analysis of the complex static-semantics is a large and complex program by itself (reports indicate a front end size between 18,000 and 120,000 lines of code). A compiler front end only dealing with the functions of the analysis phase is essentially independent of the target machine. Unfortunately, the distance in semantic level between the output of the compiler front end and the compiler back end is quite large. Splitting a compiler back end into an expander and a separate code generator is therefore often worthwhile. The expander contains a number of algorithms that are target-machine independent. It can be parameterized on target dependencies. Retargeting the compiler is

then an operation involving a redesign and reimplementation of the code generator solely.

Furthermore, splitting the compiler into a separate front end, a separate middle end and a code generator keeps the programs manageable. A clear functional distinction between compiler front end and compiler back end reduces the complexity of the compiler design. Furthermore, it eases retargetability since the intermediate program representation is essentially target-independent.

The Ada— compiler was designed with a strict separation between front end and back end in mind. Furthermore, since we decided to use an existing code generator (the code generator of a variant of the PCC (the Portable C Compiler), the separation of the back end into a separate expander and a code generator was obvious. The structure of the Ada— compiler is given in figure 3.

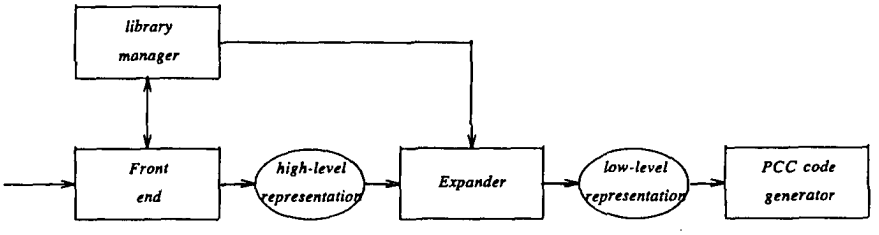


Figure 3. *Structuring of the Ada— compiler*

The input to the analysis phase is a sequence of compilation units in one or more source files. The front end is applied to each compilation unit. It scans the source program, informs the program-library manager that it intends to compile a certain library-unit or subunit. The output of the front end is an intermediate program representation, a linearized tree structure, which is passed to the expander. The expander computes the storage requirements and the access functions for all objects involved and generates a program representation in the form of a tree with low-level operations. The expander generates two external representations of the input program representation. The first is structurally the same as the input, the representation is annotated with addressing information. The second is a low-level tree-structured program representation. The annotated high-level intermediate representation is stored into the library for (potential) later use in compilations of other units. The low-level tree-structured program representation is fed into the intermediate-code improver after which it is translated by the code generator into target assembly code. The final passes are an object code improver and an assembler. (In figure 3 the intermediate-code improver, the code generator, and the peep-hole optimizer are taken together as *PCC code generator.*)

3.2.1 Compiler front end architecture

Elements in the design of the Ada— compiler front end are discussed in chapter 4. Functionally spoken, the front end of the Ada— compiler can be split up in several phases and parts:

- a driver program;
- a scanner/parser;
- a context handler;
- a static-semantics analysis phase, containing:
  - a declaration processor;
  - an overload resolver;
  - a statement checker.

A short functional description of each of the constituent parts is given below. A schematic overview is given in figure 4.

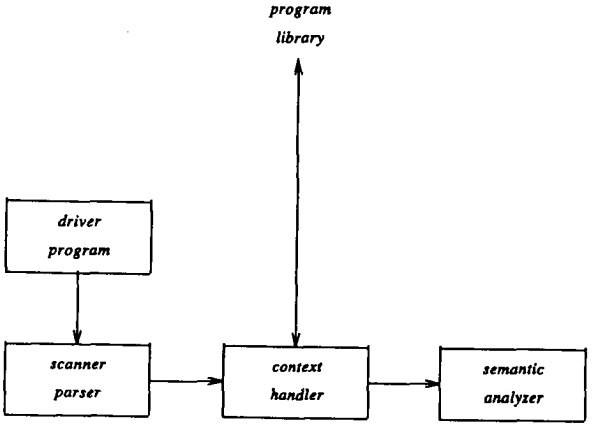


Figure 4. *Structure front end compiler*

The driver program

The driver program is the compiler's interface to the program library and the host operating system. The front end of our compiler processes source files (i.e. *compilations*) on the basis of compilation units, one at a time. At the start of the processing of a compilation unit the library is informed that an attempt is made to compile a certain compilation unit. After the compilation unit is processed, the driver calls for the

subsequent passes† of the compiler. Following the complete back end processing, the driver informs the library manager whether the attempt was successful or not.

Scanner/parser

For the construction of scanners and parsers tools are generally available. Most available compiler descriptions ignore the problems involved in the construction of a scanner or a parser.

In the Ada— compiler the parser constructs an - in-memory - parse tree in a single scan from left to right over the source text. During the construction subtrees are handed over to the semantical analysis module to perform static-semantics analysis. At the end of the parse a complete annotated parse tree is available in memory.

Context handler

The context processing phase processes *with* and *separate* clauses and it establishes the visibility of the elements in package *standard*. The phase requests the intermediate program representations of the units for which the name is mentioned in a *with* or a *separate* clause from the program-library manager.

Static-semantic analyzer

The function of the static-semantic analyzer is to check that the program does not violate the context conditions of the language. It performs a number of functions:

1. declaration processing and symbol table building. In order to allow efficient identifier look-up, declarations are entered in a symbol table structure;
2. identifying applied occurrences of identifiers by their corresponding defining occurrences. The process implementing this identification falls into two separate parts: looking-up identifiers according to the scope and visibility rules of the Ada language and second, resolving overloading whenever necessary;
3. syntactic disambiguation. The parsing phase maps several syntactical ambiguous constructions onto a single unambiguous construction. A well-known example is

   A. B (C)

   which may stand for a type conversion, an array reference, a function call, a procedure call, an entry call or a slice;
4. type checking;
5. evaluating *static* and *universal* expressions;
6. handling generics.

Furthermore, a vast amount of smaller and larger checks have to be performed. To mention a few:

---

† The word *pass* is usually associated with a single scan through the program or its representation. In our case a *pass* does not necessarily operate in a strict left-to-right or right-to-left fashion.

- verification that the left-hand side of an assignment statement is an object that can be assigned to;
- verification that aggregates containing an *others* choice only appear in certain contexts;
- verification that subprogram parameters with mode *in* are not of a limited private type;
- verification that body stubs appear only at the outermost level of compilation units. They are named by use of identifiers, not by use of operator symbols.

Most of the checks can be performed in a straightforward, though ad hoc, fashion. Type checking and overload resolution are generally impossible in a single scan from left to right through the source program.

The output of the front end is an AST, called the *DAS tree*. This tree is self-contained. It contains all information of the original source program; in particular, it contains all declarative information such that no separate symbol table has to survive between passes or even between compilations.

3.2.2 Compiler back end architecture

Several elements in the design and implementation of the compiler back end are discussed in chapter 5. The function of the compiler back end is to translate the high-level intermediate program representation into target code. The driver program invokes an instance of the compiler back end for each compilation unit. A back end for an Ada language compiler can be split up in several parts:

- an expansion phase (the *expander*), performing:
  - storage allocation;
  - lowering the semantic level of the program representation;
- intermediate-code optimization (optional);
- a code generation phase (the *code generator*), performing:
  - instruction selection;
  - target register selection and allocation;
- peep-hole optimization (optional).

In the Ada— compiler the back end is split up into four passes. The first pass is the expander. The second pass is an intermediate-code optimizer. In this pass the low-level tree structure is re-arranged and common sub-expressions are determined. The third pass is the code generator. The final pass is the peep-hole optimizer. The last three passes are all borrowed from the ACE-EXPERT compiler family on our local system; they will not be discussed further. The structure of the Ada— compiler back end is given in figure 5.

The input to the expander is the high-level tree-structured intermediate program representation. Its output is an equivalent program in a low-level program representation.

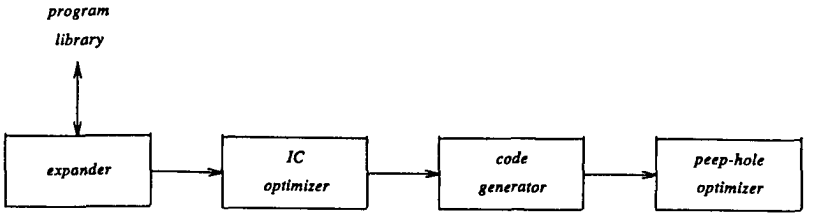


Figure 5. *Structure back end compiler*

Storage allocation, i.e. the compile-time registration of storage requirements for objects at run time highly depends on the representation chosen for objects and their descriptors at run time. In our compiler we use the so-called *doublet model*

Code is generated on a procedure-by-procedure basis. Code for nested procedure declarations is linearized, all forms of static procedure nesting are removed. Code for a procedure is generated in a single walk through the procedure.

For each expression that appears in the source program the high-level intermediate representation which is generated by the front end is transformed into a low-level intermediate representation. The essential difference between the two representations is that the operations that are on source level in the high-level intermediate representation, are on target level in the low-level intermediate representation. Furthermore, in the low-level representation all operands are expressed by their access code in terms of low-level operations whereas in the high-level representation the operands are expressed by the defining occurrences of their identifier.

Code for statements and control structures is linearized.

3.2.3 Run-time support

The Ada— compiler generates code for a real target machine. A number of operations requires run-time support. In particular, the support for complex operations as *equality test* on structured values and *default initializations* for record components requires a certain amount of run-time support.

Run-time support is also required for the implementation of the Ada tasking model. The support provided by the tasking supervisor of the Ada— compiler is discussed in detail in chapter 6.

3.2.4 The program library

The Ada language requires a program library manager for the support of separate compilation. The program library contains compilation units in an intermediate form.

The design and implementation of the Ada— compiler program library is discussed in chapter 7.

3.3 The literature on Ada implementations

In spite of the fact that currently a number of validated compilers for the Ada language exists, there exists descriptions for only a few of them. Once every few months, the *Ada Information Clearing House* publishes an overview of the validated compilers.

Compilers for which some form of technical description (either for the complete compiler or for some parts of it) exist and which are often referred to, are:

- The Charette Ada compiler. The Charette Ada compiler is a compiler for preliminary Ada which was designed and implemented as a co-project between a group at Carnegie Mellon University (CMU) and Intermetrics. The compiler was one of the very first compilers on which a number of papers appeared. The main descriptions of this six-pass compiler can be found in [Sherman-80] (the front end) and [Rosenberg-80] (the compiler back end). The compiler ran under TOPS-10 and was targeted to a VAX 11/780.

- The NYU Ada compiler/interpreter. The NYU Ada compiler interpreter is meant as an *operational semantic definition* of the Ada language. Written in SETL (a very high-level set-oriented language), it is one of the very few compilers for which sources are available [Dewar-83]. The compiler was developed on a VAX under VAX/VMS. Currently, versions of the compiler run on a variety of systems.

- The Karlsruhe Ada compiler. The Karlsruhe Ada compiler is developed at the university of Karlsruhe. Its development was sponsored by the German department of defense. A number of papers on (parts of) the compiler has appeared. It has been the basis for the commercially available SysTeam compiler. The compiler was developed on a Siemens 7000 system. The target systems are a Siemens 7000 and an M68000 instruction-set architecture.

- The European root compiler. Within the context of the multi-annual programme in the field of data processing, the commission of the European communities has supported the development of a portable Ada Root compiler. The aim of the project was to achieve a highly portable production-quality Ada compiler that was made available on a wide range of computers. The compiler was developed by a consortium consisting of Alsys/CII Honeywell Bull/ Siemens. Currently, both Alsys and Siemens have validated Ada compilers based on this root compiler.

- The AIE Intermetrics compiler. As part of the AIE, the *A*da *I*ntegrated *E*nvironment, Intermetrics has developed a compiler for the Ada language. The AIE was an APSE design, developed under contract of DoD. The compiler is hosted on a 370 architecture and targeted to the same instruction-set architecture.

- The Ada+ compiler. The Ada+ compiler is a compiler for a superset of a subset of the Ada language. The implemented language includes generics, it excludes tasking, however. The compiler is being developed at CMU as part of the Spice project. The

compiler is targeted to Perq hardware.

- The ALS compiler. The ALS compiler is an Ada language compiler developed by Softech as part of the ALS. The ALS, the *A*da *L*anguage *S*ystem, is an APSE which was developed by Softech under contract with DoD. Simpson [Simpson-82] gives a brief description of the ALS Ada language compiler. The compiler is hosted on a VAX and targeted to the same instruction-set architecture.
- The FSU compiler. The FSU Ada compiler is a compiler which was developed under contract by a group at Florida State University. The compiler is developed on a Cyber system and is targeted towards a Z8000 architecture. It has recently been validated.
- The DDC compiler. The DDC compiler is a compiler which was developed by a group at Dansk Datamatik Center. The compiler was developed as part of a project sponsored by the Commission of the European Communities to develop the PAPS, the *P*ortable *A*da *P*rogramming *S*upport environment. One of the goals of the compiler project was to construct a highly portable and rehostable front end.
- The BreadBoard compiler. The BreadBoard compiler is a compiler which was developed at Bell laboratories.
- The York Compiler. The York Ada Workbench compiler was developed at the university of York. It was one of the very first available compilers for a subset of the Ada language. The compiler was validated August 1986. Some technical details are briefly mentioned in [Wand-87].
- The Ada— compiler. This compiler is the subject of this thesis.

The NYU Ada compiler/interpreter consists of a two pass front end and an interpreter. Dewar et al [Dewar-80] discuss some elements in its design. The output of the front end is a high-level list-oriented program representation which is passed directly to the interpreter for interpretation. The size of the front end was about 18,000 lines of SETL.

The Charette Ada compiler is implemented as a six-pass or seven-pass compiler. The compiler front end consists of two passes, the back end of five passes. (Rosenberg [Rosenberg-80] considers the front end as a single pass, Sherman et al [Sherman-80] describe a two pass front end.). The intermediate code is a T_{col} variant.

The Karlsruhe Ada compiler consists of a front end, generating the intermediate language DIANA. The back end of the compiler consists of two separate parts. A first part maps the DIANA programs onto programs in the low-level intermediate language AIM. A second part of the back end produces machine code.

Intermetrics [AIE-82] has made public the B-5 specifications of the whole AIE compiler. The front end of the compiler consists of two passes, the back end consists of several phases which are logically split up in two major phases.

The structure and implementation of the Ada+ compiler is discussed by Barbacci et al [Barbacci-85]. The front end of the compiler consists of two distinct phases: syntactical and semantical analysis. The back end of the compiler also consists of two phases, post-

semantical analysis and code generation. The phases communicate through a high-level internal representation of the source program.

Simpson [Simpson-82] gives a brief description of the ALS Ada language compiler. The compiler has the common front end/back end architecture. DIANA is used as a language for the high-level intermediate program representation.

Wetherell et al [Wetherell-82] discuss the BreadBoard compiler. The BreadBoard compiler is a four-pass or five-pass compiler, depending on what is counted as a pass. The guiding principles in its design are (source: [BreadBoard-82]): *Keep it simple* and *Do a minimum of work.* The compiler uses DIANA as the language for the high-level intermediate program representations, the language C is used as a target language.

A global design of the DDC compiler is given by Bundgaard et al [Bundgaard-82]. The compiler consists of no fewer than seven passes. Each pass is a complete executable program on its own, reading its input sequentially and producing the output sequentially. The compiler design, and hence the number of passes, has been strongly influenced by the limited amount of memory available in the host computer system. The original goal, which was not met, was to have the compiler implemented on a machine with 64 Kb program space.

The York compiler has a front end that consists of five passes; the back end is a single program. Two rather shallow descriptions are given of the architecture of the York compiler [Wand-82], [Briggs-83a].

The descriptions on the FSU compiler remain restricted to descriptions of the compiler front end and descriptions of the tasking supervisor [Baker-82a], [Baker-82b], [Baker-84], [Baker-85], [Baker-86a], and [Baker-86b].

4. The Ada— compiler front end

4.1 Introduction

In this chapter some topics on the design and constructions of the compiler front end are discussed. Due to the large number of papers on front-end topics a significant part of the chapter is devoted to the discussion of these papers. Attention is given to the literature, to some practical issues in overload resolution, to implementation strategies for scope and visibility rules and to some practical issues as appearing in the development of our front end.

In our compiler model the front end of a compiler for the Ada language performs the following functions:

- it reads the input Ada source program text;
- it performs all syntactic and non-syntactic (static-semantic) checks on the input program as required according to [LRM-83];
- it handles declaration and instantiation of generic program units;
- it generates an external intermediate program representation, a flattened tree-structure. This tree structure is suitable for further processing by a compiler back end and by other tools.

The structure of the compiler front end is given in figure 6. The *driver program* is the primary user interface to the compiler. Input to the driver is an entire compilation which may exist of several compilation units. The driver program calls the scanner/parser with a compilation and the compilation option as its parameters.

The scanner/parser breaks the compilation into the individual compilation units. We have used LEX [Lesk-75] for the construction of a scanner. The parser is LALR (1) based [Aho-85]. It is generated by VEYACC, a locally extended version of YACC, the parser generator on the UNIX operating system. VEYACC [Groeneveld-86] supports a restricted class of L-attributed grammars and is furthermore equipped with a decent error-repair capability. Error repair is based on continuation functions with an extension to improve the behaviour of certain classes of simple errors. The parser specification is about 850 syntax rules.

The function of the parser is to decompose the program into a tree-structure based on the syntactical description. The parser produces one abstract syntax tree per compilation unit. The design of the parser and the applied error-repair technique are discussed in section 4.4. The parser calls the scanner as a subroutine; the latter reads the source text of the compilation. At the end of parsing a single compilation unit, the parser has produced a complete abstract syntax tree for that unit.

During the parsing, the parser hands subtrees to the next phases of the front end. A subtree representing a *with* clause or a *separate* clause is handed over to the *context handler*. This handler requests the loading of the intermediate program representation of the units involved.

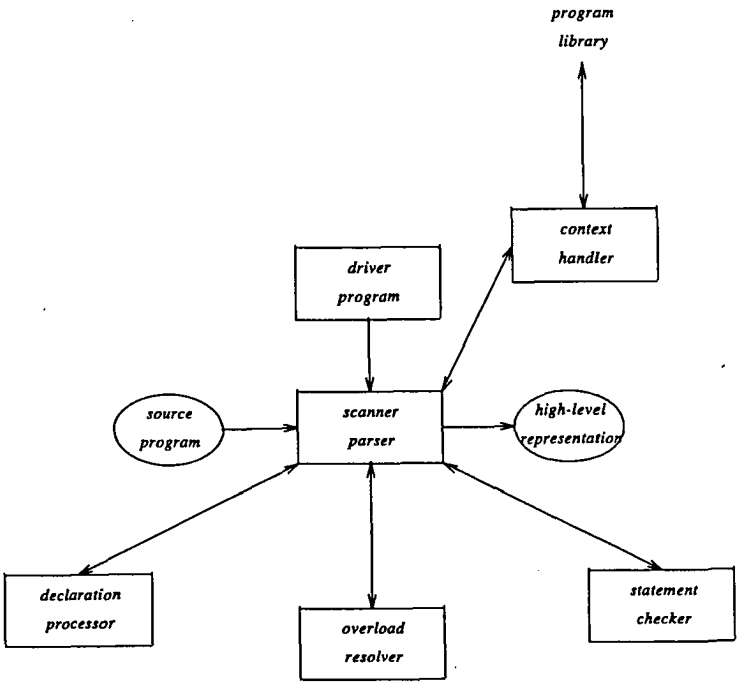


Figure 6. *Compiler front end structure*

A subtree representing one or more declarations is handed over to the declaration processor. This processor checks the validity of the declarations and enters them in the intermediate program representation (which also acts as a dictionary or symbol table).

A subtree representing an expression is handed over to the overload resolver. It is well-known that for overload resolution and type checking two scans over the expression tree are required. The first one is a bottom-up scan through the tree. In this scan information on the leaf types and the operators is propagated up into the tree. The second one is a top-down scan which is used to restrict the set of operators and operands such that each operator and each operator is identified uniquely. The overload resolver is driven by recursive tree walkers; it makes *4* scans over each subtree:

1. in the first scan the tree is traversed top-down;
2. the second scan is the bottom-up scan of the overload resolution algorithm. For each identifier the set of possible defining occurrences is computed and information on the

leaf types is propagated up the tree.

3. the third scan is the top-down scan of the overload resolution. For each sub-expression the type is determined and the operators are identified;
4. finally, the fourth scan performs a bottom-up scan again. Its function is to do some checking, to deal with static and universal sub-expressions and to bring the tree structure into a canonical form.

Details of the implementation of overload resolution and identifier look-up are given in sections 4.4 and 4.5 respectively.

A subtree representing a statement is handed over to the statement checker for checking constraints enforced by the context conditions. These checks include the verification that e.g. the range of a *case* expression is completely covered by the values of the *case labels*. A large submodule in the statement checker is the generic expander. A subtree, representing a call to a generic unit, is handled by this generic expander. Only a subset of the *generics* construct is implemented; this subset is defined such that the instantiation of both a generic specification and its body can be performed by the front end. Details of the handling of generics are discussed in section 4.6.

The output of the whole front end is an annotated abstract syntax tree, ready to be used by the compiler back end or by other tools. This tree, the *DAS tree*, is based on DIANA and other similar high-level intermediate program representations. Its structure is discussed in section 4.3.

4.2 A brief review of front end descriptions

Most descriptions of front ends for Ada language compilers remain relatively vague. This is not surprising, it remains difficult to describe the functionality or the constructions of a program between 18,000 and 150,000 lines of code in a few pages.

A number of papers have appeared in which (partial) descriptions of a front end for an Ada language compiler is given. We shall restrict ourselves here to papers and descriptions of whole front ends. Papers on particular issues in the design and implementation of front ends are recalled when the particular issues are discussed.

The front end of the NYU Ada compiler and interpreter is split up into two separate passes. Furthermore it contains a module to handle library-retrieval and library-update functions. The first pass does syntactical analysis. Only programs without syntactical errors are passed to the second pass, the semantic analyzer. The semantic phase of the NYU-Ada compiler consists of three phases: an initialisation phase, a phase in which the tree is built, and a phase performing semantic actions. The last phase also converts the program tree into a bindable form. The whole front end consists of about 18,000 lines of SETL code.

Goos et al [Goos-80] discuss the front end of the (original) Karlsruhe Ada language compiler. The parser module consists of three parts: a lexical analyzer, a pragma handler and a syntactical analyzer. Syntactical analysis is done by a LALR (1) parser. Driven by the reductions, an intermediate representation of the program is constructed, a tree-

structure known as AIDA† [Goos-80]. The semantic analyzer is concerned with type checking, control of visibility of identifiers and the resolution of overloaded occurrences of operator and subprogram symbols. An early version of the semantic analyzer was based on a preliminary, incomplete, version of the Formal Definition of Ada [FormDef-80]. It was partly obtained by rewriting the functions of the formal definition by hand into subprograms and functions written in the Ada language.

In a later stage two important modifications were made to the Karlsruhe compiler. First of all, a new intermediate program representation, DIANA [DIANA-83], was designed as co-production between the Karlsruhe group and a group from Tartan Labs Inc. Second, attribute grammars were used as the formalism for the description of the static semantics. Drossoupoulou et al [Dross-82] discuss the use of an ordered-attributed grammar for the formal description of a semantic analyzer. The text of an attributed grammar for the semantic analysis of Ada is given in [Uhl-82]. Attributed grammars proved to be a very comfortable tool for the specification of static semantics. "*..Because all language properties are expressed over the attribute dependencies, and nothing is said about passes, tree scans or function calling hierarchy, changes are very easily made. The 'specifier' concentrates on the concepts of the language reflected in the attributes...*" Nevertheless, the complete specification of the static semantics in terms of an ordered attribute grammar took about 20,000 lines. With the available GAG system [Kastens-83], a semantic analyzer was indeed generated from the attributed grammar specification. The generated semantic analyzer consisted of a Pascal program, taking up to *1.6 Mb*, processing about 1.5 lines of Ada source text per second on a Siemens 7000 system. A front end of the Karlsruhe compiler, containing a hand-written translation of the GAG generated semantic analyzer, consisted of about 120,000 lines of Ada source code, divided as given below [Dross-83].

lexical analysis	*syntax analysis*	*semantic analysis*	*DIANA generator*	*support modules*
3,000	*5,000*	*65,000*	*10,000*	*45,000*

The front end uses over 800 Kb. It processes about 8 to 10 lines a second [Dross-83]. This front end has been the basis for the SysTeam compiler, a validated and commercially available compiler.

Sherman [Sherman-80a] discusses a front end for preliminary Ada written in Simula. The front end was derived from a prototype implementation of the *red* language and is written in Simula. (The *red* language was the contribution of Intermetrics to the DoD language design competition.)

Apart from a semantical analyzer, the front end consisted of a context-free parser (also written in Simula) centered around a LALR (1) parser. Central to the design of the

† AIDA stands for *An Intermediate Description of Ada Programs.*

semantic analyzer is the use of Simula classes to represent grammar productions and Simula class instances to represent nodes in the derivation tree. Edges in the derivation tree are represented by Simula reference values, i.e. pointers to class instances. Semantic checking is performed during a scan through the derivation tree. The semantic analyzer verifies that the tree is consistent with the language semantics, it deals with issues to be discussed in section 4.5. The output of the semantic analyzer is a tree-structure, a variant of the T_{col} family.

Sherman reported that the compiler front end consisted of about 900,000 characters of Simula source code. It ran on Tops-10 and Tops-20 systems. It had a processing speed of about 2 to 3 source lines a second.

The B5 specifications of the AIE compiler [AIE-82] are fairly detailed. They give insight in how some of the problems are solved. The front end operates in two phases, a lexical/syntactical analysis phase and a semantical analysis phase. The lexical/syntactical analysis phase reads the source of a compilation unit and produces an abstract syntax tree; the semantic phase performs semantic analysis and transforms the abstract syntax tree into a DIANA tree.

Sherman's work was done at Carnegie-Mellon University (CMU). The front end was used as a front end for the so-called *Charette* Ada compiler [Rosenberg-80].

The front end of the Ada+ compiler is implemented as a single program. During syntactical analysis a tree is constructed. The semantical analysis phase does an ordered scan over that tree, annotating it with semantic information. Each grammar production is turned into a tree node; each tree node is handled by a specialized routine. Calls to the overload-resolving and the generic-handling modules are scattered throughout the semantics code. As the compiler scans the tree, it creates type and symbol blocks to represent various Ada entities and intertwines these blocks with the parse tree. In order to keep the complexity of the semantical analysis phase manageable, all code for overload resolution and handling generics is put in separate modules. Some attention is paid to the problems occurring with type checking and overload resolution. Generic instances are handled by a post-semantic phase. This phase does an ordered scan over the annotated tree, looking for subtrees that represent generic instances. It copies the bodies of generic templates to create bodies for their instances.

Simpson [Simpson-82] gives a rather brief description of the ALS Ada compiler. The compiler was implemented as part of the ALS, the Ada Language System, developed under DoD contract. In only three pages the author gives a rather brief overview on the semantic analysis phase of the ALS compiler. The output of the front end is a DIANA-based intermediate program representation. Syntactical analysis is by a LR(1) rather than a LALR (1) type of parser. Description of the semantic analysis phase is split up in a number of separate points: processing of the context, name resolution, overload resolution, the evaluation of static expressions and statement checking. Final activity of the semantic processing unit is the generic expansion. The description relates to Ada-80 rather than to ANSI Ada, a separate subsection is devoted to the transition to (at that time) preliminary ANSI Ada.

Baker et al [Baker-82a], [Baker-82b] give a rather detailed description of a one-pass syntax directed front end for the FSU Ada compiler. This FSU (Florida State University) compiler was developed on a Cyber system and was targeted towards a Z8000 microprocessor architecture. It has been recently validated. The construction of the front end is unorthodox in the sense that the whole front-end processing is done in a single left-to-right scan. Parsing is by an LALR (1)-type parser, extended with the capability of disambiguating on reduce/reduce conflicts with the help of *semantic* information. Most other compilers have the work in the front end separated in a context-free parsing module, delivering a tree-structure, and a semantic analysis phase, decorating the tree-structure. In later subsections we discuss the one pass approach to type checking.

Wetherell et al [BreadBoard-82] discuss the BreadBoard compiler. The first pass of the compiler [Wetherell-82a] performs lexical analysis, syntactical analysis and pragma handling. For syntactical analysis a LR (1) rather than a LALR (1) parser is used, primarily because of the availability of a generator for LR (1) parsers [Wetherell-81a]. A small intermediate pass, the *pre-semantic* pass [Wetherell-82b], checks the output of the parser, an abstract syntax tree, for conformance to the structural rules that are beyond the expressive power of a context-free grammar. The output of this pre-semantic pass is a DIANA-based intermediate program representation.

Quinn [Quinn-82] discusses the second pass of the BreadBoard compiler. It reads the DIANA abstract syntax trees, analyses them and generates decorated trees.

A global design of the front end of the DDC compiler is given by Bundgaard et al [Bundgaard-82]. The *test front end* compiler, as the DDC compiler is called, consists of no fewer than five passes. Each pass is a complete executable program on its own, reading its input text sequentially and producing the corresponding output text sequentially. The compiler design, and hence the number of passes, has been strongly influenced by the limited amount of memory available in the host computer system. An original goal was to have the compiler implemented on a machine with a 64 Kb program space. The first pass of the compiler performs lexical and syntactical analysis; the next three passes concentrate solely on checking context conditions; finally, the fifth pass implements the top-down part of overload resolution and expansion of generic instances. This last pass was estimated to be the largest one, about 9400 lines of Ada code. At an early stage of the development it already became clear that none of the passes was capable of building the entire intermediate program representation in main memory and performing all checks that require symbol table information. Symbol table information must therefore survive between passes. Furthermore the amount of symbol table data is likely to depend on the size of the compilation unit; hence the symbol table itself cannot even be represented in full in main memory and some software paging mechanism was required for the symbol table

Bundgaard [Bundgaard-85] gives a retrospective of the final DDC front end. He reports that the implementation of the whole compiler in a 64 Kb address space turned out to be impossible. A version of *almost* the whole compiler which was encoded in some highly space-efficient P-code variant, did fit in 64 Kb.

The construction of a highly portable and rehostable front end was one of the original design goals of the DDC compiler. This goal seems to have been achieved, the front end has been rehosted on various architectures, including a VAX-11/780, a Honeywell DPS6 and a NOKIA MPS 10 computer.

The York Ada Workbench Compiler was one of the very first available compilers for a subset of the Ada language. Only a few papers are available on the global structure of the compiler [Wand-82], [Murdie-83] and [Briggs-83a]. The York compiler is a 6-pass compiler (or a 3-pass compiler, depending on the reference); the first five passes comprise the front end. The first pass is a top-down syntax analyser; the next three passes carry out analysis of declarations, scopes and names; the fifth pass performs type analysis and the remaining machine-independent semantic checks. The tree-structured intermediate representation is called the AIR, the Ada Intermediate Representation. Wand [Wand-87] reports that the total size of the front end is about 57,000 lines of C code.

4.3 Intermediate program representations

The use of intermediate languages in program translations is getting increasing attention in recent years. Modern high-level languages and modern implementation techniques (translator-writing tools) often require the design of multi-pass compilers. The choice of an adequate intermediate language is important, especially to people engaged in the design and implementation of portable software.

One of the characteristics of high-level intermediate languages is that they are derived from the source language; they usually are based on an annotated Abstract Syntax Tree. This AST is enriched with attributes; context dependencies that are implicit in the original source program are made explicit in the intermediate representation. It is often possible to generate a functionally equivalent source program or even to regenerate the original source program.

A major characteristic of low-level intermediate languages is that they are often derived from the union (or the intersection) of a number of real target architectures. A low-level intermediate language often defines some hypothetical target machine.

A high-level intermediate program representation is often a by-product of the design of a modular compiler. On the other hand, low-level intermediate program representations are more related to portability through the definition of a hypothetical computer.

Several papers have been published on intermediate program representations. Chow and Ganapathi present a bibliography on intermediate languages in compiler construction [Chow-83]. They state that the choice of intermediate code is important to people engaged in the design of compilers, code generators and optimizers, and, more specifically, to those concerned with building portable software. Their survey does not seem complete, however.

Teller [Teller-80] discusses requirements for intermediate languages and characteristics of both high-level intermediate languages and low-level intermediate languages. He states that the suitability of an intermediate language from the compiler writer's point of view is primarily related to how well the intermediate language conceptually fits into the overall

compiler design. A question to be asked to an intermediate program representation is which information contained in the source language program is still available. In particular, to what extent it is possible to regenerate the original source program or to generate a functionally equivalent source program.

Compilers for the Ada language often employ some high-level intermediate program representation between the compiler front end and the compiler back end. A rationale for a high-level intermediate representation was already given in chapter 3.

A low-level intermediate program representation is often used as well. This low-level representation is then used to generate target code. A rationale for having a low-level intermediate representation is to increase the retargetability by separating (almost) target-independent parts (generating the low-level intermediate representation) from the code-generating parts. Such low-level representations usually take the form of code for a hypothetical target computer or it is in the form of low-level expression trees.

This approach differs from the one made in many implementations for Pascal-like languages. In implementations for the latter only a low-level intermediate representation is used to obtain a higher degree of retargetability.

Programming environments, as built and envisaged for Ada, contain tools such as syntax-directed editors, librarians and linkers that require the existence of some high-level intermediate program representation. In the latter part of 1980, AIDA was used for the Karlsruhe Ada compiler. At the same time a T_{Col} variant was used at CMU for their Ada compiler. Groups of the two institutions met by invitation of SofTech and during a one-week design session a new intermediate program representation for Ada was designed that blended the best of the two original representations. This new intermediate representation was named DIANA. DIANA was meant as a standardized intermediate program representation. Several compiler-design efforts have been using DIANA as an intermediary notation, including Burroughs, Intermetrics, Karlsruhe, Rolm, SofTech and Bell labs.

On the other hand, a large number of people in the Ada community does not believe in DIANA as intermediate program representation. J. Ichbiah, the spiritual father of Ada, publicly stated † that he rejected DIANA as being too heavy to use. Indeed, the use of DIANA leads to rather large intermediate representations for programs. Wilcox [Wilcox-85] reports that the external intermediate representation for an average sized Ada program takes up to 300 Kb in the Rational environment.

The AIR, the Ada Intermediate Representation, used in the York compiler is broadly similar in aims and implementation to DIANA. Nevertheless, the two representations have very little in common. Descriptions of the AIR, its design, and its application to separate compilation are discussed extensively by Briggs [Briggs-83b]. It turns out that the ideas behind the AIR and our high-level intermediate program representation have some elements in common.

---

† The statement was made during a keynote address at the *Ada in Use* conference (Paris 1985).

4.3.1 The DAS tree: design considerations

The high-level intermediate program representation for the Ada— compiler is called the *DAS tree.* The DAS tree is used as intermediate program representation between the compiler front end and the compiler back end. It is essentially a tree-structured intermediate representation, although it can also be viewed upon as a network of weakly typed nodes.

The DAS tree was designed at a very early stage of the development of the compiler. Emphasis in its design was put on easy reading and writing *parts* of the representation from and to external files. At the time the intermediate representation was conceptualized, processing was done on a PDP-11 with 64 Kb address space. It was clear from the very beginning that it was impossible to have the whole intermediate representation for a program in memory. Therefore, each compiler phase was conceptually modelled as a tree walker; reading tree parts, processing these parts and writing these parts out again. Only data essential for the compilation of the whole of the remaining program was kept in memory.

The limitations of the operating environment have influenced the design of details of the DAS tree considerably. It is, however, our strong belief that almost any intermediate representation (in which all context dependencies of the source program are made explicit) suffices. It is merely a matter of finding a balance between *finding* data in the intermediate representation and *recomputing* data (either from the intermediate representation or from the source program). Designing an intermediate representation may be oriented towards a certain form of processing. In our representation we have chosen to have a number of trivial items recomputed, whereas for items for which recomputation of the value is hard, the value is computed once and stored in the tree-nodes.

The intermediate program representation is tree-structured. Its structure closely resembles a parse tree. The internal form is pointer-based, the external representation is linearized in a prefix form. The attributes of the tree nodes, both the number of attributes and the domain of their values, are determined by:

- the kind of processing to be done, i.e. is the whole tree-structure in memory or are only subtrees in memory;
- a relation between attributes and their values to the data that is being computed during static-semantic analysis. Our representation represents an abstraction of both the syntax and the semantics of a compilation unit;
- engineerings trade-offs that have to be made in deciding which attribute values are stored and which are re-computed;
- the need to be able to generate a source program which is functionally equivalent to, though not necessarily literally the same as, the original source program.

The Ada language supports separate compilation. Intermediate program representations of several units have to coexist in memory. The external intermediate program representation has to contain an identification of the unit it belongs to. In our approach the data comprising a compilation unit in a program library is (part of) the intermediate

program representation. Our choice has been inspired by DIANA; the intermediate representation contains all definitions from the source program. No separate dictionary† is maintained; the intermediate representation itself plays a role similar to that of a dictionary in current compiler technology. The purpose of our dictionary mechanism (or better: name list, section 4.5) is *only* to speed up the process of binding an applied occurrence of an identifier to the corresponding set of visible defining occurrences. Once such a binding has been established, references are maintained between nodes in the intermediate representations of the various representations involved.

The intermediate representation for a given compilation unit is in itself a complete tree-structured representation of that compilation unit. The structure of the source program is directly encoded in the tree-structure, static-semantic information is encoded as attribute values (either as plain values or as references to other nodes).

Essentially, for each declarative item or statement in the compilation unit the DAS tree of the compilation unit contains a node or a subtree. A package

```
package P is
   ....
end P;
```

is represented by a tree rooted by a node labelled *PACKAGE*. This node contains a (reference to a) list of declarations contained in the package.

A statement, e.g.

```
A := B + C
```

is represented by a tree rooted by a node labelled *ASSIGN*. Operations which can take the form of a function call are represented in a standardized form, i.e.

```
B + C
```

is represented as if it were written as

```
"+" (B, C)
```

We have chosen for a uniform functional representation for all calls to overloadable functions and operators. A rationale for this representation is that from a purely syntactical point of view no difference can be made between predefined operators such as e.g. the "+" on integers, and user-defined operators, e.g. a "+" on arrays, records or matrices. The code generator has some extra work to do with the former, i.e. in-line

† a *dictionary* is a table in which defining occurrences of identifiers are recorded and which is organized so as to speed up the process of finding defining occurrences with a given identifier. The term *symbol table* is also often used. We prefer the term *name list* for our implementation since what is maintained in our implementation is simply a mapping from identifiers to nodes in the tree. The sole purpose is to speed up the process of finding defining occurrences with a given identifier.

encoding of the operation. On the other hand, the latter is translated to a function call, so a functional representation is obvious.

Furthermore, for each parameter of a function call the binding with the corresponding formal parameter is made explicit. The schematic tree-structure for the above statement is given in figure 7.

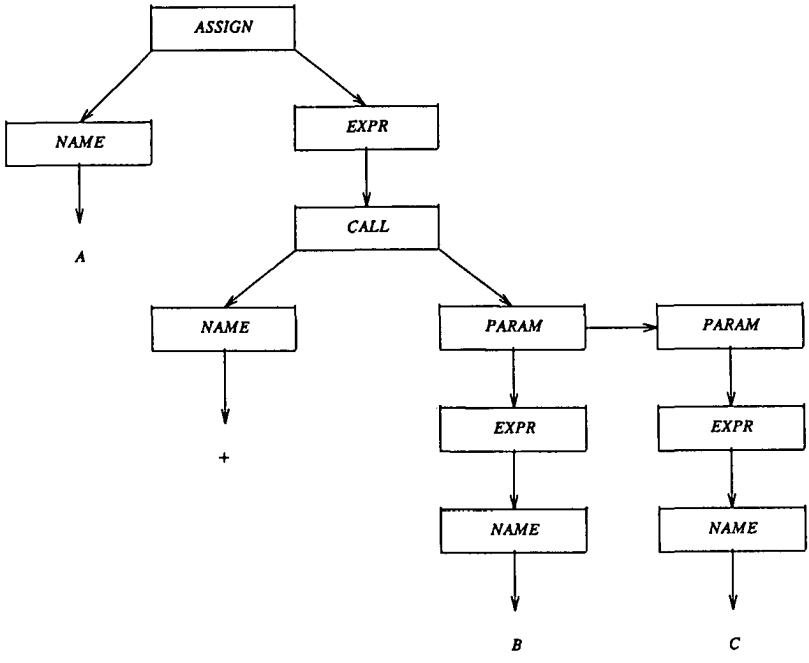


Figure 7. *Tree for A := B + C*

The pointers that establish the tree-structure are called *structural links*

Context dependencies are expressed by attribute values. Simple values and flags are denoted by plain values. *Links* are used to represent dependencies on other nodes. These links are called *semantic links*. Consider the example of the statement

A := B + C

again, where A, B, and C are variables of integer type. The annotated tree of the statement, in the context of

type *integer* : ...

and a specification

function "+" *(left, right: integer)* **return** *integer;*

is given in figure 8. (In this figure semantic links are indicated by representing the targets of the links between { and }).

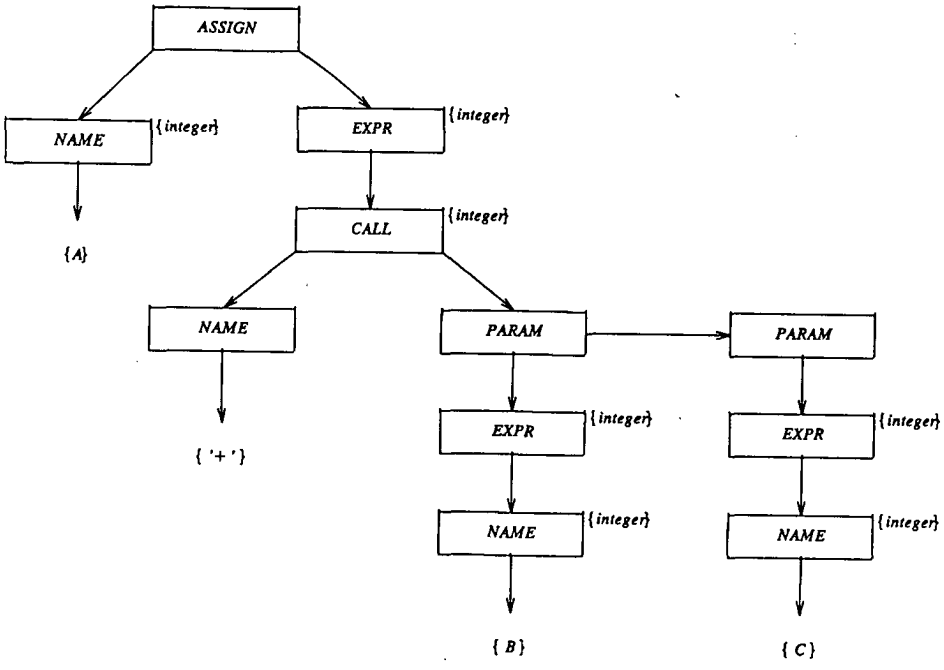


Figure 8. *Annotated tree for A := B + C*

For example, the semantic link to the expression type appearing in the *EXPR* node, contains a reference to the defining occurrence of a type of the expression. In this case a pointer to the definition of *integer*. The *EXPR* node contains a structural link to its primary.

It can be argued that several of the semantic links that appear in the intermediate representation of a given constructions are superfluous. In the example above the types of the various expressions can be computed from the types of their primaries. The choice between storing a piece of data or recomputing that piece of data is obviously an engineering decision.

A requirement for having the intermediate tree representation act as a dictionary is that for each Ada entity a *single definition* point exists such that each use of the entity refers to this unique defining occurrence. Attributes at the definition point record the information about the entity. This principle conflicts with entities in the Ada language that have more than one declaration for the same entity; some examples:

- incomplete type definitions;
- private type definitions;
- subprogram declarations and their body;
- package declarations and their body.

For any case where there is more than a single declaration for a given entity, the first appearance is defined to be *the* defining occurrence of that entity. Consider the example of a package *p* with a contained procedure specification for *q*;

```
package p is
  procedure q;
end p;

...

package body p is
  procedure q is
    ....
  end q;
end p;
```

Any reference to *p* is interpreted as a reference to the package *specification*, independently of the context. Similarly, any reference to the identifier *q* is interpreted as a reference to the procedure specification with that identifier.

An incomplete type (and similarly a private type) definition always acts as the definition point for the type. Consider the example of the incomplete type *a*.

```
type a;
type a is array (1 .. 10) of integer;
b : a;
```

The reference from the object descriptor for *b* is a reference to the incomplete type definition *a*, as is the reference in the *NAME* node in

```
b (i) := 1
```

During the parse the parse tree is built; during semantic analysis this tree is annotated. The top node of the parse tree is, however, not the root node of the whole tree in memory.

Each library unit is compiled in the context of a package *standard.* This package *standard* contains definitions for the common types and operations. The intermediate tree representation of the library unit is, preceded by the subtrees of the library units specified in the context specification, linked as an ordinary declaration in package *standard.* Therefore, the node describing package *standard* is always the top node of the whole tree.

A similar situation arises with secondary units and with subunits. On processing a secondary unit, the intermediate program representation of the parent unit has to be loaded. When the parent unit is a library unit, it has to be linked into the declarations for package *standard*; when it is not, its tree has to be linked into the tree of its parent unit.

4.3.2 Implementation and external representation

The internal program tree is built using a C structure for the description of the nodes. Each node is implemented as a member of a variant record; for each node type a discriminating value is defined. Currently about 85 node types exist. Symbolic representations of some discriminating values of nodes are:

SUB	*subprogram*
SUBBODY	*subprogram body*
EXPR	*expression*
NAME	*applied occurrence of identifier*
AGGREGATE	*aggregate value*
ENUMTYPE	*enumeration type*
ASSIGN	*assignment statement*
CALL	*function call*
NAMEDASS	*named parameter association*

Tree nodes contain, apart from a discriminating value, an identification to be used for the representation of references to the node in external representations of the tree-structure in which the nodes appear. The identification is an order number of the node within the nodes used in the compilation unit.

Since the intermediate program representation is a pointer structure implementing a tree, a pointerless external representation is possible. In the Ada— compiler the external representation is a prefix-encoded tree structure. Although a pointerless representation leads to a fairly compact way of storing the intermediate representation there is still a 1 : 4 ratio between the sizes of the source program and the external intermediate program representation.

Consider e.g. again the example:

A := B + C

The ASCII-encoded external tree representation becomes (omitting needless details):

[ASSIGN

```
[ NAME ]                    (A)
[ EXPR
  [ CALL
    [ NAMEDASS
      [ EXPR
        [ NAME ] ] ]        (B)
    [ NAMEDASS
      [ EXPR
        [ NAME ] ] ]        (C)
    [ NAME ]                (+ )
    ] ] ]
```

To eliminate possible ambiguities the external representation of any subtree is enclosed in square brackets.

Each node is externally represented by a series of values. Each value represents either a plain value (e.g. a discriminating value) or a semantic link to another node in the tree. Since semantic links may refer to nodes in trees for other units, a two-dimensional addressing scheme is required. Each semantic link is represented by a pair

```
(cun, node)
```

where *cun* (compilation unit number) identifies the compilation unit by a number and *node* is an order number identifying the node in the compilation unit. An ASCII-encoded intermediate program representation for the statement

```
A := B + C
```

in an environment containing suitable declarations for the entities involved, is given below. In this (generated) intermediate representation an attribute is indicated as

```
attribute_name => attribute_value
```

```
[ ASSIGN
 [ NAME                      -- left operand
  types => (  1,  4)         -- predefined integer type
  fentity => ( 115,  6) ]    -- encoding for A
                             -- node 6 in unit 115
 [ EXPR                      -- right operand
  exptype => (  1,  4)
  [ CALL                     -- function call
   types => (  1,  4)        -- result type
   [ NAMEDASS                -- subtree for parameter
    parname => (  2, 343)    -- right hand formal of "+"
```

```
[ EXPR                  -- corresponding actual
  flags => static
  exptype => ( 1, 4)
  [ LITERAL             -- primary
    littype => ( 1, 4)
    litkind => 1
    value => 1 ] ] ]    -- close three subtrees
[ NAMEDASS              -- left hand formal of "+"
  parname => ( 2, 344)
  [ EXPR
    exptype => ( 1, 4)
    [ NAME
      types => ( 1, 4)
      fentity => ( 115, 4) ]]]
[ NAME                  -- function being called, i.e. "+"
  types => ( 2, 341)
  fentity => ( 2, 341) ] ] ] ]
```

4.3.3 Reading and writing intermediate trees

Given the fact that the external intermediate program representation is a prefix-encoded representation, reading it or generating it is fairly straightforward.

To write an intermediate program representation to an output file, a depth-first, left-to-right walk is made through the pointer structure implementing the tree. The nodes are written on the prefix encounters; first the node itself and then the subtrees rooted by this node are written. The representation of a subtree is terminated by an end marker. (Notice that this slightly differs from the examples in the previous subsection where externally represented tree-structures were enclosed in square brackets.) The routine *put_node* takes care of generating an external representation of the node itself including its attribute values such as semantic links; the tree-writer only linearizes the tree; the procedure implementing the tree-writer is called *put_tree.* Furthermore, a procedure *put_list* is defined. The structure of the *put_tree* subprogram is schematically:

```
...

case discriminating_value (node) is
...
when EXPR =>
        put_node (node);        -- print the node
        put     (g_primary (node)); -- print it's primary
        put_mark;               -- print an end marker

when NAME =>
        put_node (node);        -- print the node
        put_mark;               -- print an end marker
```

```
when CALL =>
        put_node (node);        -- print the node
        put_list (g_param  (node)); -- print the actuals
        put     (g_caller (node)); -- print the caller
        put_mark;               -- print an end marker

...
```

The approach made for reading external tree representations is similar. A procedure *read_node* reads the individual nodes, the actual tree-structures are read by procedures *read_tree* and *read_list*. The structure for *read_tree* schematically reads:

```
....

current_node := read_node;
case discriminant_value
...
when EXPR =>
        s_primary (node, read_tree); -- read the primary
        check_mark;                  -- check on end marker

...
when NAME =>
        check_mark;                  -- check on end marker

when CALL =>
        s_params (node, read_list);  -- read the parameters
        s_caller (node, read_tree);  -- read the caller
        check_mark;                  -- check on end marker
....
```

Readers and writers for the tree are generated from a pseudo-formal tree description, similar as for the AIR of the York compiler.

4.4 Parsing and error repair

Context-free descriptions in reference manuals for programming languages are often made ambiguous to improve readability. The context-free description in the Ada language reference manual [LRM-83] is no exception to this rule. A consequence is, however, that the syntactical specification as given in [LRM-83] is not suited as a basis for the constructions of a parser.

A remarkable number of people has either published a syntactical description (LL (1) or LALR (1)) or has criticised the description in the Ada language descriptions. Examples can be found in [Bonet-81], [Cole-81], [Persch-81], [Smith-84], and [Fisher-84]. Although the description in e.g. [Fisher-84] is a LALR (1) based description of the Ada language grammar, it is not really suitable as a basis to construct a front end for an Ada language compiler.

When constructing a parser for a language as large as the Ada language is, two issues have to be addressed. First, the degree of compatibility between the grammar in the reference manual and the actual syntactical description is to be investigated and the accuracy of the syntactical description is to be determined. Second, the problem of recovering from syntactical errors should be addressed. These issues will be discussed in the following subsections.

Another issue to be addressed is the design of the parser output. In our compiler the parser builds a complete tree in memory and passes subtrees for semantical analysis to a semantical analysis module. When the parser finishes its job, the whole tree for the compilation unit is available. Tree building as a by-product of bottom-up parsing is well understood.

The parser used in our compiler is generated by an extended version of YACC. YACC is the parser generator available under UNIX. We found some extensions to YACC indispensable. A first extension was in the form of a preprocessor. The preprocessor allows the use of a limited form of L-attributed grammars. The well-formedness criteria were chosen such that a simple single stack implementation of the generated parsers is possible. The preprocessor, described by the author in [Katwijk-83], was later incorporated in VEYACC, our extended version of YACC. The second extension improved the error-repair capabilities of generated parsers.

4.4.1 Compatibility in and accuracy of descriptions

It can be argued that the degree of compatibility between the grammar that is used as basis for an implementation and the reference grammar should be as high as possible. Context-free descriptions are often designed to be readable; this readability should be affected as little as possible. On the other hand, it can be argued that a syntactical description should be reasonably accurate. A number of constraints *can* be expressed in terms of a simple context-free grammar. If reasonably possible these constraints should be expressed that way. Unfortunately, expressing constraints usually affects the readability. Therefore compromises have often to be made between the accuracy of the grammar on one hand, and readability and compatibility with the context-free description of the reference manual on the other hand.

A particular problem is the ambiguity in the context-free syntactical specification. Parser generators as e.g. YACC allow syntactical specifications to be ambiguous. They apply either user-specified or default *disambiguating rules* to *resolve* such ambiguities. From a user's point of view it is then often difficult to determine what the language is which is recognized by the parser. Nevertheless, it turns out that in some cases the design of a syntactical description is eased when some ambiguities are allowed.

In this subsection we shall elaborate on accuracy of syntactical descriptions.

The description of the *name* constructs is a good example. The description of this construct as given in [LRM-83] is highly ambiguous. Several authors [Wetherell-81b], [Baker-82a] have noticed this and commented on it. The relevant part of the ambiguous specification of [LRM-83] is given below.

name ::=
    simple_name
  | *character_literal*
  | *operator_symbol*
  | *indexed_component*
  | *slice*
  | *selected_component*
  | *attribute*

simple_name ::=
    identifier

prefix ::=
    name
  | *function_call*

indexed_component ::=
    prefix (expression {, expression})

slice ::=
    prefix (discrete_range)

It is impossible to distinguish between a slice, an indexed component and a function call, only based on this description. The practical approach is, of course, to rewrite the grammar such that the offending constructs are factorised to a single syntactic category. The decision as to whether the construct is a slice, an indexed component, or a function call, is then postponed until the semantical analysis phase. During semantical analysis, attributes of the types and entities involved are identified and a decision can be made. A unified, unambiguous, specification as proposed by many authors, e.g. [Fisher-84], is given below.

name ::=
      simple_name
    | *character_literal*
    | *operator_symbol*
    | *indexed_component*
    | *selected_component*
    | *attribute*

```
simple_name ::=
                identifier

prefix ::=
                name

indexed_component ::=
                prefix (expression {,expression})

selected_component ::=
                prefix . selector

selector ::=
                simple_name
              | character_literal
              | operator_symbol
              | all

attribute ::=
                prefix ' attribute_designator

attribute_designator ::=
                simple_name [(universal_static_expression)]
```

This description is unambiguous and fairly compatible with the description in [LRM-83]. Nevertheless, it is inaccurate. Consider the name construct

```
A (B). 'c'
```

denoting the selection of a character literal *c* in a unit *A (B)*. Whatever the meaning of the prefix *A (B)* might be, the result can never be a prefix acceptable in a unit selection. It is clear, just by looking at this construct, that selection of a character literal in this context is never legal. A parser based on the above specification accepts nevertheless such a construct and semantical analysis has to deal with the detection of the violation of the well-formedness rules.

It is often straightforward to make a description more accurate. A syntactical specification of the name construct, based on the one applied in our compiler, is given below. The specification is ambiguous; the disambiguating rules of YACC are taken into account in its design.

```
name ::=
  x_name
| x_name attribute

x_name ::=
```

```
x_name ( expression {, expression} )
| x_name attribute
| x_name . all
| x_name . identifier
| lim_name
| lim_name . all

lim_name ::=
  identifier
| lim_name . identifier
| lim_name . character_literal
| lim_name . operator_symbol
```

The construct

```
A (B). 'c'
```

is syntactically rejected. Furthermore, this syntactical description provides support for the distinguishing between the *BASE* attribute and other attributes. It is easy to check whether the appearance of a *BASE* attribute is legal or not. Since it can never occur as last attribute, the attribute specification in

```
x_name attribute
```

can never be a *BASE* attribute.

By carefully designing the syntax of the various constructs, performing certain well-formedness checks becomes much easier. Similar situations arise in many other places.

4.4.2 Syntactical error repair

One of the consequences of moving a constraint to the syntactical specification is that the checking of this constraints is done by a parser. Violation of the constraint leads to a syntactical rather than to a semantical error. Syntactical error recovery or repair† is often rather crude. It affects the parse tree that is being built to the effect that semantic processing on the subtree involved is (almost) impossible.

Most proposals or descriptions with regard to error recovery or error repair consider a two-stage scheme: first some form of local recovery/repair is tried; if that fails, then recovery/repair is tried on a more global level. The local recovery/repair is often based on the following steps:

1. try to insert a symbol before the current symbol. If that does not solve the problem try step 2;

---

† By *repair* we mean transforming the not yet processed input such that the parse may continue. By *recovery* we mean transforming the parse stack and the, not yet processed, input such that parsing may continue.

2. try to delete the current input symbol. If that does not solve the problem try step 3;

3. try to replace the current input symbol by another one that does not cause an error.

This technique is a simplified version of the technique described by many authors, e.g. Graham et al [Graham-79b].

When local recovery/repair fails, another error routine is invoked. This latter routine takes more context into account in a second attempt to recover from the error or to repair it.

A number of compiler front-end descriptions contains a rather detailed description of error-repair and recovery strategies applied. A brief review of these methods and techniques is given. Then a more detailed description of our syntactical error handling is presented.

Dewar et al [Dewar-80] describe the error repair in the NYU translator. This translator is meant to be used for training purposes and, consequently, a large effort has been put into a good syntactical error detection and recovery. A two-stage mechanism is used. First the state of the (LALR) parser is backed up to the point following the shift of the token preceding the token that caused the error. Often in this process default reductions have to be undone. For local error handling a technique similar to the one described above is applied. If local repair fails, an error-recovery procedure is invoked. The first form is *scope recovery* in which it is tried to match on some important syntactic bracket (an *end* symbol for example). The second form is *list recovery*. The assumption is here that the error occurs in a list-like structure such as a *list of statements* or a *list of declarations*. Tokens are skipped from the input until a token is encountered which can start an item from the *nearest* list on the parse stack.

The Ada+ front end uses a recursive-descent parser [Barbacci-85]. Each production in the language is parsed by a function which returns the corresponding part of the syntax tree. The applied error-recovery technique is a form of follow-set recovery [Stirling-85], which is customary for recursive-descent type parsers. Whenever an error is detected, lexical elements are discarded until some member of a *recovery set* is found. The recovery set is passed as a parameter to the function. The recovery set is a follow-set which can be computed dynamically.

The error-handling scheme in the FSU Ada compiler can "*...be considered unusual in its simplicity*" [Baker-82a]. As usual with LALR-based parsing techniques, the two stage scheme is applied. Coming across an error, first an attempt is made to repair the error locally. If that attempt fails, a second attempt is made using *error productions* [Aho-85] to recover from the error.

Some compilers, e.g. the BreadBoard compiler and the ALS compiler, employ full LR(1) parsing rather than LALR (1) parsing. Full LR parsing has an advantage over LALR (1) type of parsers: when coming across a syntactical error, no default reductions have been performed. Therefore, no forms of backing up erroneous reductions have to be attempted. The parse configuration in which the error is detected is therefore more precise.

Wetherell [Wetherell-82a] reports that the BreadBoard compiler uses heuristics in its error handling. The error-correction heuristic assumes that one and only one error has caused the current parsing problem. The error is assumed to be caused by an erroneous symbol *deletion, replacement* or *insertion.* These assumptions are tested by reversing the error process. The process is repeated after putting a symbol of the stack back into the input. If no correction is found within a certain limit of the point where the error was detected, the original offending symbol is simply deleted and the correction process restarted from the point of deletion.

The error recovery applied in the AIE compiler is also a two-stage method. The first stage is the local attempt to repair the error locally, the second stage, which is only applied if the first stage fails, is based on error productions.

The error repair applied in the various versions of the Karlsruhe compiler is a *continuation* based method; theory of such methods is developed in Karlsruhe [Röhrich-80]. A description of continuation based techniques is given below.

The front end of the Ada— compiler also employs the two-stage method. If local repair fails then an attempt is made for a more global repair. Since we had some experience with continuation-based methods in previous compiler projects, it was decided to use a continuation-based method for our Ada compiler as well. In a later stage a local repair based on the steps described earlier was added to the implementation.

The basic philosophy behind continuation-based methods is that on the detection of a syntactical error a *continuation string* is constructed. A continuation string is a string such that the concatenation of this string to the already processed prefix of the input forms a valid sentence. The assumption that a continuation string can indeed be constructed is based on the prefix property which is a property valid for LR type parsers.

Let

v_i

be a parser-defined error† in

$x_1, x_2, \ldots, x_n, v_i, \ldots, v_k$

Determine *some* continuation string $c_1, \ldots, c_l$ such that

$x_1, \ldots, x_2, \ldots, x_n, c_1, \ldots, c_l$

is a correct sentence. Since

$x_1, \ldots, x_n$

is a viable prefix, at least one such string exists.

The next step in the repair process is to determine a set of pairs of indices

$\{(h, j)\}$

† a *parser-defined error* is the symbol causing the parser to detect a syntactical error.

such that for each pair

(h, j)

it holds that a replacement of

$v_i, \ldots, v_j$

by

$c_1, \ldots, c_h$

results in a viable prefix

$x_1, \ldots, x_n, c_1, .. c_h, v_{j+1}$

Error repair is done by selecting a pair

(h, j)

and performing the replacement of

$v_i, \ldots, v_j$

by

$c_1, \ldots, c_h$

Since in general more than a single repair is possible, a selection has to be made from a set of possible repairs. In [Katwijk-84b] experience with this kind of error repair in the DAS compiler is reported. Experiments were made with selection criteria based on the number of deleted symbols, the number of inserted symbols and the number of symbols that could be parsed after repair before the subsequent error was found. As can be expected, the most influential factor is the number of symbols that is accepted before running into the next error.

The problem of (mechanical) construction of a postfix is not treated here. A detailed description of formalisms behind algorithms is given discussed by Röhrich [Röhrich-80]. In our environment the construction of the tables, required for the postfix construction, was done by hand, though aided by a set of tools [Toet-84a]. When VEYACC became available, the computation was done by the parser generator in parser-generation time.

Repair based on the computation of continuations has advantages: it is both simple and it gives reasonable results. On the other hand error repair, even based on continuations, does sometimes produces strange results. The first problem we came across was that LALR (1) type of parsers often do a number of parse steps (default reductions) on meeting an erroneous symbol. A second problem, particular to continuation-based methods, is that only a single continuation is taken into consideration.

To demonstrate the practical consequences of delaying error detection due to default reductions is given below. Consider the program fragment:

```
begin
   null;;    -- second ; erroneous
end;
```

The error message generated by an early version of our compiler was

'loop null; end loop *inserted'*

This somewhat surprising result is completely due to default reductions. In this particular case the (simplified) syntactic structure causing this behaviour is:

```
stat :   opt_id loop_or_block opt_id
       | some_other_statement

opt_id :  ϵ
       | identifier

loop_or_block:
       loop_start loop_stat
      | declare ...
      | begin ...

loop_start :
       ϵ
      | while ...
      | for ...

loop_stat :
       loop statements end loop;
```

On the start of the recognition of strings produced from *stat* with a *parser defined error* in the input stream, first reductions are made of the rules

```
   opt_id :  ϵ
and
   loop_start : ϵ
```

Preventing default reductions entirely requires a full LR (1) parser rather than an optimized LALR (1) one.

A limitation of continuation-based methods is that they only take into account a single continuation. Having a single continuation often leads to strange forms of repair. Repairs that could have been made according to a different continuation, simply by inserting or replacing one or two symbols, are completely overlooked. Consider

```
X : (my_range) of my_subtype;                    -- array omitted
```

In our situation, the continuation which is computed does not contain the symbol **array**. Since the symbol *identifier* is in the computed continuation string the error is repaired by either inserting *identifier* or deleting the (symbol. Both choices lead to an avalanche of

subsequent errors. This particular case caused us to investigate local error handling and to add the local error handling that was described earlier. In a case like this the locality of the error will cause the error to be repaired by the local error handler.

The combination of local error handling with a continuation-based method was applied in our compiler and is fairly satisfactory. On the other hand, it should be clear by now that most forms of syntactical error repair are based on some form of educated guess. Although elaborate forms of guessing may sometimes lead to an acceptable repair, the general expectation of the quality of the repair should remain modest. It seems rather better to concentrate on user interfaces preventing errors than on error-repair techniques.

4.5 The handling of static semantics

It seems that the current state of technology does not allow formal methods to be widely used in language specification and compiler implementations. An exception is perhaps formed by attributed grammars. Drossoupoulou et al [Dross-82] report on the use of an attributed grammar to describe the semantic phase of the Karlsruhe Ada compiler. A prototype semantic analyzer was generated from the attributed description. The attributed specification for the Ada programming language, i.e. the formal description of the semantic analyzer was published in [Uhl-82]. The generated front end was rewritten, however, to get an acceptable performance.

Various authors report on a pseudo-formal approach where *a* formal description of the Ada language is used as a basis for the semantic analysis. In [Goos-80] a description is given of the use of [FormDef-80] as basis for a semantic analysis and an example is given on how the resulting semantic analyzer is derived from the formal description. As a demonstration we give in figure 9 a procedure from the formal definition and in figure 10 an in Ada transformed procedure as given in [Goos-80]. The Ada Formal Definition [FormDef-80] is based on denotational semantics. It is given in an undefined applicative language with an Ada-like syntax.

```
function CHECK_DECL (decl_s:   TREE;
        global, local:  S_ENV) return TREE_ENV is
begin
  if IS_EMPTY (decl_s) then
    return TREE_ENV (EMPTY (decl_s), local);
  else
  declare
    tree_env:     constant TREE_ENV :=
                      CHECK_DECL_S (HEAD (decl_s),
                                global, local);
    incremental:  constant S_ENV   := ENV (tree_env);
    local1:       constant S_ENV   := NEST (local,
                                    incremental);
    global1:      constant S_ENV   := NEST (global,
                                    incremental);
    global2:      constant S_ENV   :=
```

```
                    UPDATE_CURRENT_UNIT (global1,
                                         incremental);
    tree_env1:    constant TREE_ENV :=
                  CHECK_DECL_S (TAIL (decl_s), global2, local1);
  begin
    return TREE_ENV (PRE (TREE (tree_env),
                  TREE (tree_env1)), ENV (tree_env1));
  end;
  end if;
end CHECK_DECL_S;
```

Figure 9. *Function from formal definition*

A hand-translated version in the Ada language of this function is given in figure 10.

```
separate (MAIN_PROGRAM. SEMANTICS. SEMANTIC_RULES)
procedure CHECK_DECL_S (DECL_S:  in AI_TREE;
            ENV:      in SE_ENV;
            LOCAL:    in SE_ENV) return SE_TREE_ENV is
PROC_RES:    SE_TREE_ENV;
TREE_ENV:    SE_TREE_ENV;
LOCAL2:      SE_ENV;
ENV2:        SE_ENV;
TREE_ENV2:   SE_TREE_ENV;
begin
  if DECL_S = AI_EMPTY then
    PROC_RES := (CTREE => AI_EMPTY, CENV => LOCAL);
  else
    TREE_ENV := CHECK_DECL (P004 => AL_HEAD (DECL_S), P003 => ENV,
          P002 => LOCAL, P001 => SC_UNDEFINED_LIBDESC);
    LOCAL2   := TREE_ENV. CENV;
    ENV2     := SE_NESTING_OP (ENV => ENV, ENV2 => LOCAL2);
    TREE_ENV2:= CHECK_DECL_S (DECL_S => AL_TAIL (DECL_S), ENV => ENV2,
          LOCAL => LOCAL2);
    PROC_RES := (CTREE => AL_PRE (TREE => TREE_ENV. CTREE,
                        LIST => TREE_ENV2. CTREE),
            CENV => TREE_ENV2. CENV);
  end if;
  return PROC_RES;
end CHECK_DECL_S;
```

Figure 10. *Formal definition and manual translation*

The resemblance between the two fragments is clear.

The semantical analysis of the DDC compiler is based on a home-made partial formal description [Bruun-81].

Semantical analysis in the BreadBoard compiler is also based on a formal definition for Ada [FormDef-80]. The experience reported by Quinn [Quinn-82] is worthwhile mentioning. According to Quinn: " ... *The design of Pass 2 is based on the Formal Definition of the Ada Programming Language (FD) .., a denotation semantic definition of the static and the dynamic semantics of Ada. The FD is written in an undefined applicative language that has an Ada like syntax. ... The FD is, unfortunately, incomplete. Checks for some constructs are omitted entirely, and numerous functions -particularly in the symbol table package- are referenced but not defined. The first version of Pass 2 was a hand translation of the FD into the language C with rudimentary implementations of the missing function bodies. During the development of Pass2, many errors in the FD were discovered and Pass2 was revised accordingly* ...". A similar experience was reported by Goos et al [Goos-80]. This coincided with our experience, we started with the formal definition and moved slightly, though steadily, away from it.

Contrary to the handling of syntactical errors, the handling of semantical errors is a topic ignored in most front end descriptions. The function of semantical error repair is to allow semantical analysis to proceed and to avoid an avalanche of errors.

Johnson et al [Johnson-82] discuss the approach in handling semantical errors taken in the York Workbench compiler. An attempt is made to move to a more formal treatment in the repair of semantical errors. The approach is presented by giving useful concepts and rules. A special tool, DIAPER, *D*iagnostic *I*nterpreter for *A*da *P*rogram *ER*rors, is introduced. This tool has three major component functions:

1. it is a storage and retrieval system for full error message texts; the user can select and filter individual reports;
2. it is a tutor, providing help from a simple database that contains potted explanations of language rules and terms in the error messages;
3. it is an interpreter, offering interactive diagnosis for certain kinds of errors such as those which have to do with scope, overloading or type representation. The interpreter includes a collection of experts, each of which specializes in a particular aspect of the language.

The 'experts' used are:

- a scope expert, allowing a user to start from a chosen point, such as the location of a reported error, and to discover what is in the scope, what is visible and what is hidden. The scope and visibility rules of Ada are rather complex, justifying a tool providing the user information on what is effectively visible or hidden. The scope expert operates on a so-called *current* scope, a scope to which the commands of the user are considered to be related.
- a type resolution expert, providing a convenient notational framework within which questions of errors like

line x: 'return' expression meaningless

can be put. Just as the scope expert operates on a current scope, the type resolution expert operates on a notion of current expression.

We believe that more research is required on the subject of systematic recovering from errors in the semantical analysis.

In the following subsections several topics will be discussed in more detail: practical problems in overload resolution and the implementation of the complex scope and visibility rules of the Ada programming language and the handling of generics. Even for the non-Ada compiler writer a more thorough discussion of these issues is worthwhile.

4.5.1 Overload resolution and operator identification

In the Ada programming language several kinds of entities, e.g. enumeration literals, character literals, operator symbols, function identifiers and procedure identifiers, can be overloaded. A *designator* is said to be overloaded if it has more than a single meaning (section 8.3 [LRM]) according to the scope rules of the language. In order to determine the meaning of the designator, the context in which the designator is used plays an important role. The needs for overloading as well as possible solutions are discussed at length in the Ada rationale [Rationale-79] (section 7.5). The Ada programming language is by no means the first programming language providing overloading; most programming languages allow some form of overloading for the usual arithmetic operators. e.g. "+" and "*". ALGOL 68 [ALGOL-68] allows operators to be overloaded with user-defined operators with the same indicant. PL/I allows overloading of subprograms with its *generic* facility.

A consequence of allowing overloading designators is that type checking in expressions and the identification of the operators cannot be performed in a single bottom-up scan through the expression. Instead, two scans, a bottom-up scan followed by a top-down scan are required to identify all operators and to determine the type of all intermediate constructs. This requirement is seen in other languages, type checking and operator identification in ALGOL 68 requires also a bottom-up followed by a top-down scan.

Since the appearance of preliminary Ada, the rules for overload resolution have significantly changed. In the Ada versions of 1979 [Ada-79] and 1980 [Ada-80] overload resolution had to be resolved using two attempts: first overload resolution was tried without taking into account declarations imported through use clauses; if the attempt failed, overload resolution had to be applied for a second time: now with taking into account declarations made visible through "use clauses". Thus the same algorithm had to be applied twice. In the process of revising the Ada language, ultimately leading to the ANSI standard [LRM-83], the rules for overload resolution were considerably simplified. No difference was made any more between definitions made visible through use clauses and definitions made visible through open scopes. The simplification of the rules made that overload resolution had to be applied only once. Furthermore, the rules themselves were simplified.

The theoretical aspects of overloading and overload resolution in the Ada language are well known. The emphasis in this subsection is therefore on practical aspects of overload resolution. The structure of the subsection is as follows: the problem is illustrated by a small example and a survey of the literature available is given. Then the outline of a straightforward bottom-up/top-down algorithm is given. Special cases, in particular handling universal types and operators as e.g. "=", are discussed next. Finally a short discussion is devoted to some implementation aspects of the implementation of overload resolution in the Ada— compiler.

The problem and a survey of the literature

The problem of overload resolution is probably best explained by a simple example. Let

function *f (a, b : T1)* **return** *T;*
function *f (a, b : T1)* **return** *T1;*
function *f (a, b : T2)* **return** *T2;*

be functions with identifier *f*. The type of the expression

$$f(e_1, e_2)$$

cannot be deduced from the function call itself; the context in which it appears must be taken into account. If the context requires the expression to be of type *T2*, the *f* in the expression can be identified as being the one from the third definition. With the operator identified, the types of the parameter expressions can be deduced.

Particularly in the early days of Ada the subject of overload resolution has received much attention. Ganzinger et al [Ganzinger-80] are one of the first to describe an algorithm. In their paper a three-pass algorithm is presented and proven to be correct. The algorithm uses three scans, a top-down scan, a bottom-up scan and a top-down scan again. The basic algorithms for the bottom-up and the top-down scan will be described later.

Penello et al [Penello-80a] eliminate the initial top-down scan from the algorithm of Ganzinger et al; only the bottom-up/top-down combination is left. Their algorithm and its correctness is discussed in some more detail in the next subsection.

Janas [Janas-80] presents an improved version of the original algorithm of Ganzinger et al. He presents conditions under which complete subtrees can be neglected during the second top-down scan. Local stabilization of those subtrees is fairly simple to detect. Penello et al [Penello-80b] give some more simplifications.

Persch et al [Persch-80] give a thorough description of overload resolution. The two-scan algorithm is given again and it is proven to be correct. The proof is quite readable. It is introduced by means of an example described in terms of a simple attribute grammar. We disagree with the conclusion that the process of overload resolution must not be considered as to "*be rather time consuming*", opposed to the conclusion in the Ada rationale [Rationale-79].

An optimized version of the two-scan algorithm is discussed by Wallis et al [Wallis-80]. Their algorithm has the interesting property that after completion of the bottom-up scan it

is clear whether the expression tree under consideration has a unique interpretation or not. Unfortunately, if it does not have a unique interpretation, no further information is available on the offending (sub)construct.

Belmont [Belmont-80] discusses the implementation model for overload resolution that was proposed for the AIE compiler. Dewar et al[Dewar-80] discuss issues in overload resolution as applied in the NYU Ada compiler.

A first implementation of overload resolution in our (first) DAS compiler was implemented by Bal [Bal-82]. Essentially a two-scan algorithm was used. Our current implementation deviates in many places from this first implementation. Nevertheless, it still contains traces of the work of Bal.

Cormack [Cormack-81] is worried about the time and space requirements of the various algorithms; he presents some optimizations on the original three-scan algorithm. The optimizations are, however, more improvements of the programming than algorithmic improvements.

Baker [Baker-82a], [Baker-82b] claims that overload resolution can be performed in a single, bottom-up, scan. In a later subsection his approach will be discussed. It is our claim that his algorithm requires two scans as well.

Briggs [Briggs-83a] presents an (incomplete) solution on which the implementation in the York Workbench compiler is based. The underlying ideas seem rather ad hoc.

Barbacci et al. [Barbacci-85] present a short description of the overload resolution in the Ada+ front end. A more detailed description of the overload resolution is given by Stockton [Stockton-85]. According to this author the algorithms are based on the ones given by Baker.

Most papers cover the straightforward cases. In only a few papers some attention is given to extensions required to handle universal types and special standard operators as e.g. = (the equality operator). Our implementation model is derived independent of others. There is nevertheless a certain similarity between our method and the methods derived and implemented by others. In particular, there is similarity between our method and the method described for the Intermetrics AIE compiler [AIE-82]. Our method seems, at least from the description, somewhat simpler and more general in its approach to universal types and special standard operators that the approach given by e.g. Ganzinger.

The basic algorithm

We start with a basic algorithm for the identification of an operator in an expression. At a later stage the basic algorithm is extended to cope with the more complex cases. In the algorithms presented here some simplifications have been applied:

- only function (or operator) calls are considered;
- neither named parameter associations nor default parameters are taken into account.

Overload resolution, implying the identification of the proper operations of all nodes in an expression tree, is in two steps. The first step is a bottom-up scan through the expression

tree which restricts at each node in the expression tree the set of operators by discarding those operators whose formal-parameter types are inconsistent with the types of the actual-parameter expressions. The second step is a top-down scan through the expression tree which restricts the set of operators at each node in the expression tree by discarding each operator whose result type is inconsistent with the formal-parameter type of the corresponding formal parameter in the set of operators associated to the parent node.

Consider again the three definitions for *f*. Let

function *f (a, b : T1)* **return** *T;*
function *f (a, b : T1)* **return** *T1;*
function *f (a, b : T2)* **return** *T2;*

be functions with identifier *f*. Consider the function call

$$f(e_1, e_2)$$

Let us assume that the bottom-up scan yields the types *T1* and *T2* as possible parameter types. In the bottom-up scan through the function call the set of possible definitions for *f* remains the set with the three possible definitions for *f*.

Let us further assume that the context requires the result value of the function call to be of type *T2*. Taking this context type into account in the top-down scan reduces the set of applicable functions to exactly one, the *f* from the third definition. Given this definition for *f*, it is clear that the type of the actual-parameter expressions has to be *T2*. A top-down scan through the actual-parameter expressions can now be made to determine the types of the constituents of the actual-parameter expressions.

The bottom-up and top-down scan can be formulated in a more formal style. We follow the approach given by Penello et al [Penello-80b].

First some notations and functions have to be defined

We assume an expression tree with a node *u*.

1. *OP (u)* yields the set of operators that apply to *u* according to the scope and visibility rules;
2. *O (u)* yields the set of operators that are associated with the node.
   Initially *O (u) = OP (u)*;
3. *u. j* is used to indicate the *j*-th child of the node *u*;
4. *Res_type (S)* yields the set of return types of the operators in a set *S*. As a special notation we use

   $Res\_type^{-1}(T, S)$

   where *T* is a set of types and *S* is a set of operators for a function yielding the a subset of *S* of operators that have a result type in *T*.
5. *Param_type (S, j)* yields the set of types that occur as type of the *j-th* formal parameter of the elements of *S*. Similar to Res_type^{-1}, $Param\_type^{-1}(T, j, S)$ is used to compute

a subset from S of operators with type of the *j-th* formal parameter in a set T;

6. *Arity (S)* yields the *adicity* or the *rank*, i.e. the number of operands, of the operators in a set S.

After the bottom-up scan and the top-down scan have been made over an expression tree, the following result should hold:

BU_TD: For each node u and each child $u.j$ of u,

$$Res\_type\ (O\ (u.j)) = Param\_type\ (O\ (u), j)$$

In words: the set of result types of the operators at $u.j$ is the same as the set of parameter types required by the corresponding parameter of operations at u.

First, the algorithm is presented, then it is proven to be correct.

```
algorithm operator_identification
for an expression tree rooted with r
in a context requiring type C_t

1  let r be the root of the given expression tree
2  O (r) = OP (r) ∩ Res_type^-1 (C_t, OP (r))
3  for each other node v do
4        O (v) = OP (v)
5
6  { Actual tree traversals in two depth-first, left-to-right scans }
7
8  Visit_bottom_up (r)
9  Visit_top-down (r)
10 where Visit_bottom_up (u) =
11    for j in 1 .. Arity (O (u)) loop
12       let v = u.j  { v is j-th son of u }
13       Visit_bottom_up (v)
14       O (u) = O (u) ∩ Param_type^-1 (Res_type (O (v)), j, O (u))
15    end loop
16 and Visit_top_down (u) =
17    for j in 1 .. Arity (O (u)) loop
18       let v = u.j  { v is j-th son of u }
19       O (v) = O (v) ∩ Res_type^-1 (Param_type (O (u), j), O (u))
20       Visit_top_down (v)
21    end loop
```

The necessary initialisations can be made during the initial bottom-up scan through the expression tree.

A proof of the fact that the two passes of the algorithm are necessary and sufficient is straightforward. The result formulated in property *BU_TD* can be split up into two other

properties, named *BU* and *TD*. *BU* should hold after the bottom-up scan is performed, *TD* should hold after the top-down scan has been performed and should leave the *BU* property invariant.

BU: After a bottom-up scan has been made it holds for any node u that O (u) contains only operations with parameter type that the corresponding children of u actually can provide. For each node u and each child $u.j$ it therefore holds that one of the operations in O ($u.j$) has a result type equal to the type of the j-*th* parameter of an operation in O (u). Therefore it holds that for each node u and each of its sons $u.j$

$$Param\_type\ (O\ (u), j) \subseteq Res\_type\ (O\ (u.j))$$

TD: The operations in $O\ (u.j)$ supply types "acceptable" to at least one operator in $O\ (u)$; i.e. for each node u and each of the sons $u.j$ it holds:

$$Res\_type\ (O\ (u.j)) \subseteq Param\_type\ (O\ (u), j)$$

The bottom-up scan may be thought of as an information-collecting phase in which each parent u collects the various restrictions imposed on itself by its children. The top-down scan is then an information-distributing phase in which u informs each child $u.j$ of requirements imposed by its siblings on the parent and therefore on $u.j$ itself.

The bottom-up/top-down strategy can be proven to be correct by showing that the bottom-up scan achieves *BU* and then the top-down scan achieves *TD* while preserving *BU*.

Proof of BU. It can be proven inductively that *BU* holds after the bottom-up scan through the tree. If u is a leaf then *BU* is trivially true. If u is an interior node with K children the algorithm establishes that:

$$O\ (u) \subseteq Param\_type^{-1}\ (Res\_type\ (O\ (u.j)), O\ (u.j))$$
$$for\ i <= j <= K$$

or

$$Param\_type\ (O\ (u), j) \subseteq Res\_type\ (O\ (u.j))$$
$$for\ i <= j <= k$$

which is in fact *BU*.

Proof of TD. Similarly, *TD* can be shown to be correct, leaving the property *BU* invariant.
For leaves of the expression tree the assumption is trivially true. For an interior node u with K children the algorithm establishes that

$$O\ (u.j) \subseteq Res\_type^{-1}\ (Param\_type\ (O\ (u), j), O\ (u))$$
$$for\ i <= j <= K$$

or

$$Res\_type\ (O\ (u.j)) \subseteq Param\_type\ (O\ (u), j)$$
$$for\ i <= j <= k$$

which is in fact *TD*. What remains is to show that *BU* still holds
If

$$Param\_type\ (O\ (u),\ j) \subseteq Res\_type\ (O\ (u.\ j))$$

and $O\ (u.\ j)$ is then replaced by

$$O\ (u.\ j) \cap Res\_type^{-1}\ (Param\_type\ (O\ (u),\ j)),\ O\ (u))$$

then it must be shown that

$$Param\_type\ (O\ (u),\ j) \subseteq Res\_type\ (O\ (u.\ j)) \cap Res\_type^{-1}\ (Param\_type\ (O\ (u),\ j)),\ O\ (u))$$

Since

$$Res\_type\ (A \cap B) = Res\_type\ (A) \cap Res\_type\ (B)$$

it holds that

$$Param\_type\ (O\ (u),\ j) \subseteq Res\_type\ (O\ (u.\ j)) \cap Res\_type\ (Res\_type^{-1}\ (Param\_type\ (O\ (u),\ j)))$$

or

$$Param\_type\ (O\ (u),\ j) \subseteq Res\_type\ (O\ (u.\ j)) \cap Param\_type\ (O\ (u),\ j))$$

which is true because *BU* was initially true. Therefore, after application of line 19 in the algorithm, *BU* still holds.

An implementation model

A practical implementation of overload-resolution closely follows the algorithm given earlier. The main difference is in the application of set operators. In our implementation the set operations are non-atomic; they are implemented through iterations over the various sets. It is assumed that the input of the overload-resolving procedures is a tree.

A tree node u is characterized by:

- a component *defs*, initialised with a set of definitions that are visible and have the proper identification, i.e. $O\ (u)$;
- a component *restypes*, to contain a set of result types of the elements of the *defs* component i.e. $u.\ restypes = Res\_types\ (u.\ defs)$;

Furthermore, we assume the existence of some functions:

1. a function $Actual\ (u,\ j)$ implementing $u.\ j$,
2. *types_of*: construct → set of types

   types_of can be applied to any Ada expression. It yields the set of result types still valid for that expression or primary;

3. *has_one_element*: set → Boolean

   has_one_element indicates whether the set has exactly one element or not;

4. is_member: type × set → boolean

   is_member determines whether the type given as first parameter is equal to an element from the set given as second parameter or not;

5. *Arity*: operator → integer

   Arity yields the adicity or the rank of the operator given as parameter.

The existence of a procedure *bottom_up* is assumed. This procedure performs a bottom-up scan over the construct that is given as parameter. The procedure *bu_call* performs a bottom-up scan through a function-call node. It is given in figure 11.

```
procedure bu_call (node:  tree node) is
begin
  for i in 1 .. number_of_actuals
  loop
    bottom_up (parameter (node, i));
  end loop;

  node. restypes = Φ  ; -- i.e. the empty set
  -- implement O (u) =
  --   O (u) ∩ Param_type -1 (Res_type (O (u. j)), j, O (u))
  for o in node . defs
  loop
     if not applicable (o, node)
     then
       node. defs := node. defs - {o};
       -- remove o from node. defs
     else
       node. restype := node. restype ∪ Res_type (o);
     end if;
  end loop;
end;

where
function applicable (f:  definition;
                     node: tree node) return boolean is
begin
  if Arity (f) /= number_of_actuals
  then
    return false;
  end if;
```

```
  for j in 1 .. Arity (f)
  loop
    if not is_member (Param_type (o, j),
                      types_of (Actual (node, i)) )
    then
      return false;
    end if;
  end loop;
  return true;
end;
```

Figure 11. *The procedure bu_call*

The result of the bottom-up scan is a tree with a possibly reduced set of applicable operators. These operators are left in *node. defs* and a computed set of result types is left in *node. restype.* The set is further reduced by the application of *td_call* (figure 12). The second parameter to this procedure is the set of context types with which the construct must be solved.

```
procedure td_call (node:  tree node; ct: set) is
begin
  -- implement
  -- O (u) = O (u) ∩ Res_type -1 (ct, node)
  for o in node. defs
  loop
    if Res_type (o) not in ct
    then
      -- return type shows that this definition
      -- is not applicable
      node. defs = node. defs - {o};
      -- remove o from node. defs
      node. res_types = node. res_types - Res_type (o);
      -- remove Res_type (o) from result types
    end if;
  end loop;

  -- at this stage a single definition should be left
  if not has_one_element (node. defs)  -- i.e. empty or more
                                       -- than a single element
  then
    -- cannot identify an operator
    some_error;
    return;
  end if;
```

```
  -- now a top-down scan through the actual parameter
  -- expressions.
  o := node. defs;          -- the last one
  for i in 1 .. Arity (o)
  loop
    top_down (Actual (node, i), Param_type (o, i));
  end loop;
end;
```

Figure 12. *Procedure for top-down scan*

The procedures as presented here are not particularly efficient. As usual, a number of trade-offs has to be made between time and space complexity. In our implementation we emphasize simplicity of the implementation which usually leads to an increased time complexity. The efficiency of the algorithms depends in particular on the efficiency of set operations.

The function for performing overload resolutions on other operations, such as indexing, slicing and selection have a structure similar to the ones above. An example of overload resolution is analyzed in subsection 4.5.2.

The handling of special operators and universal types

Although the type system in the Ada language is really powerful, it is still impossible to describe the syntactical interface of all operators in terms of the language. Consider the example of the equality operator "=". This operator is implicitly *re*declared for each non-limited-private type that appears in a program. It seems not sensible to generate a new definition of this equality operator for each such definition. If possible, a compiler should maintain a single predefined operator "=", applicable to any non-limited-private type.

To deal properly with the "=" operator (and with similar other ones) the notion *universal type* is extended. In the Ada language itself universal types are already introduced for various purposes. In our model *universal types* appear both as types for formal parameters in function, subprogram, and operator definitions, and as types for actual objects and values. Universal types appearing as formal parameter type are called *formal universal types*, universal types appearing as types for objects and values are called *actual universal types*. Both classes are discussed below.

Formal universal types

A *formal universal type* is a universal type that appears as a formal parameter type in either an operator, a function or a subprogram specification. The main purpose of using a formal universal type in a definition is to specify a single operator or function that is applicable to a whole class of types: in some sense a polymorphic operator or function. Given the existence of a formal universal type *any_type*, the definition of the equality operator reads:

```
function "=" (left, right: any_type) return boolean;
```

Apart from *any_type*, the following formal universal types exist:

1. *any_vector*, used in the definition of the "&" operator (See section 4.5.3 in [LRM-83])

   function "*&*" *(x, y : any_vector)* **return** *any_vector;*

2. *any_integer*, used in the definition of some numerical operators;
3. *any_floating* and *any_fixed_point*;
4. *any_enumeration*, used in definitions for operators on enumeration literals.

The procedures implementing the overload resolution can remain essentially unchanged, although the rules for matching between types of actual parameters and types of formal parameters must be adapted. Consider

is_member (formal_type, actual_type_set)

which is used in the function *applicable* given earlier. The function yields *true* iff *formal_type* happens to be a formal universal type and *actual_type_set* contains a type *t* which belongs to the class of types indicated by the formal universal type. The rules that indicate whether or not a formal type stands for a given type are straightforward. Some examples:

1. *is_member (any_type, s)* = *true* for any set *s* that contains at least one non-limited-private type *t* (i.e. a type that has equality and assignment);
2. *is_member (any_vector, s)* = *true* yields *true* for any set *s* that contains a non-limited-private type *t* that is a one-dimensional array type.

Actual universal types

An actual universal type appears (temporarily) as type of an object or an expression. It stands for a type to be determined during overload resolution. During this overload resolution, objects and values that were marked with an actual universal type are forced to take another non-universal type. This type is determined by the context in which the construct appears.

Consider the example of a string literal *s*. The type of the string literal is completely determined by the context in which it appears. Prior to performing the overload resolution, a string literal is marked with the actual universal type *some_string*, a type indicating that the object is just any kind of string. During overload resolution the actual string type is determined from the context.

Apart from *some_string* other actual universal types can be identified as well, e.g.:

- *some_access type*, the type initially taken as the result type of an allocator;
- *some_integer*, the type initially assigned to an integer literal, implementing the type *universal_integer* from [LRM-83];

- *some_float*, the type initially assigned to a floating literal, and *some_fixed*, the type initially assigned to a fixed point literal;
- *some_aggregate*, the type initially assigned to an aggregate.

The rules for matching formal parameter types with actual universal types are straightforward. Each actual universal type simply stands for a whole class of types. An actual universal type matches a given type if the type stands for a class containing the base type of the latter type. As an example

some_integer **in** {*integer, real, colour*}

(taken from the procedure *td_call* with substitutions for the variables and parameters) yields *true*.

In the implementation no distinction needs to be made between formal universal types and actual universal types. The context in which a universal type appears is sufficient to determine its kind.

Matching actual and formal parameters
Consider

$e_1 = e_2$

in the context of the definition

function "=" *(left, right: any_type)* **return** *boolean;*

Assume that

types_of $(e_1) = \{t_1\}$
types_of $(e_2) = \{t_2\}$

Under the assumption that both t_1 and t_2 are non-limited-private types both

is_member (any_type, $\{t_1\}$*)*

and

is_member (any_type, $\{t_2\}$*)*

yield *true*. The given definition of "=" should nevertheless *not* be identified as the operator involved. An implicit additional requirement is that the types of the left and the right operand are equal.

This constraint holds for any operation specified to have parameters of some universal type: the types of the corresponding actual parameters should be the same.

To verify this requirement a slight adjustment of the overload resolution procedures is required. Our modifications to the algorithms that were given earlier are based on the work described by Schonberg et al [Schonberg-82]. Schonberg et al discuss overload resolution for numeric types. The same problems occur with functions and procedures having universal parameter types, we therefore give a generalised solution.

Consider the call

$$O\ (e_1, e_2, \ldots, e_n)$$

where O is some operator symbol.

$$\{ o_1, o_2, \ldots, o_k \}$$

is the set of operators identifying O.

Let o_i be defined as

$$o_i : t_1 \times t_2 \times \ldots \times t_n \rightarrow t_{n+1}$$

with the types $\{ t_i, \ldots, t_j \}$ of the formal parameters $\{i, \ldots, j\}$ being a formal universal type t. An observation is that an o_i may be applicable for one of the actual parameter types at such that

$$at \in types\_of\ (e_i) \cap \ldots \cap types\_of\ (e_j)$$

and

$$is\_member\ (t, \{at\})$$

yields *true*. Verification whether a particular o_i is applicable or not is therefore to be done for each at in the set

$$types\_of\ (e_i) \cap \ldots \cap types\_of\ (e_j)$$

Our implementation approach is to define for each applied occurrence of an operator symbol a set of *virtual definitions*. These definitions are based on the operator, the formal and the actual parameter types. A virtual definition is represented as a triple

$$< o_i, t, at >$$

o_i is the function under consideration, t is a formal universal parameter type appearing in the definition o_i and at is an element from the set

$$types\_of\ (e_i) \cap \ldots \cap types\_of\ (e_j)$$

such that

$$is\_member\ (t, \{at\}) = true$$

For any o_i with no parameters of some formal universal type, its definition is encoded as a virtual definition:

$$< o_i, \textbf{null}, \textbf{null} >$$

Virtual definitions are used in the bottom-up and in the top-down scan of the overload resolution. In the adapted form the function *applicable* verifies the viability of a virtual definition. In the top-down scan the function *td_call* propagates the context information through the parameters of the definition.

An outline of the modified function *applicable* is given in figure 13, followed by an outline of the function *td_call* in figure 14. It is assumed that a triple is encoded as a

record with components *function_def*, identifying the function involved, *univ_type*, identifying the formal universal type involved, and *actual_type*, identifying the actual type involved.

```
function applicable (t:    triple;
                     node: tree node) return boolean is
begin
  if Arity (t. function_def) /= number_of_actuals
  then
    return false;
  end if;

  for j in Arity (t. function_def) loop
    if t. univ_type =
                type_of (Actual (node, j))
    then
      if not is_member (t. actual_type,
                        types_of (Actual (node, j)))
      then
        return false;
      end if;

    elsif not is_member (Param_type (t. function_def, j),
                                types_of (Actual (node, j)) )
    then
      return false;
    end if;
  end loop;

  -- all parameters seem all right
  return true;
end ;
```

Figure 13. *Modified version of function applicable*

The function *bu_call* creates and inspects virtual definitions. To each *CALL* node a set of triples is associated rather than a set of defining occurrences.

If the result type of a function is a formal universal type, it is equal to that of one of its parameters. The actual result type of the call can be derived from the virtual definitions for that function in a call. If *f* is a function e.g.

```
function f (x : any_type) return any_type;
```

then the result type is completely determined by the virtual definitions that are created for it on a given call.

The use of virtual definitions is demonstrated by an example. Consider the call

```
f + f
```

in the context of the definitions:

```
function f return integer;
function f return long;
function "+" (x, y: any_integer) return any_integer;
```

The triples implementing virtual definitions for + are:

```
{ <"+", any_integer, integer>
  <"+", any_integer, long> }
```

The set of tuples indicates that there are two possibilities left. The first one calls for the *f* yielding an integer valued result, the second one for the *f* yielding values of type *long* as result.

Consider the definition of the function "+" as given above. The result type is the same type as the parameter type. Therefore, the actual result type is the actual parameter type for which the call is made.

The top-down scan has also to be adapted to the use of virtual definitions. The skeleton of the modified routine for the top-down scan through a function call is given in figure 14.

```
procedure td_call (node:    tree node; ct:   set) is
begin
  for t in node. defs loop
    if Result_type (t) not in ct
    then
      -- return type shows that this virtual definition
      -- is not applicable, it will be removed
      node. defs := node. defs -{t};
      -- remove t from node. defs
    end if;
  end loop;

  -- at this stage we expect a single element in the
  -- set of virtual definitions
  if not has_one_element (node. defs)
  then
    -- cannot identify a unique operator
    some_error;
    return;
  end if;
  -- now a top-down scan through the actual parameters,
  -- with respect to the type for which this
  -- virtual definition is "derived"
  -- let t be the single triple in the set
  -- node. defs
  t = node. defs;
  for i in 1 .. Arity (t. function_def) loop
    if Param_type (t. function_def, i) =
                                    t. univ_type
    then
      top_down (parameter (node, i), t. actual_type);
    else
      top_down (parameter (node, i),
                Param_type (t. function_def, i));
    end if;
  end loop;
end;
```

Figure 14. *Modified top-down scan*

It is assumed that the function *Result_type* yields the actual result type of the function invocation when applied to a virtual definition.

An operator which is defined in terms of formal universal types stands for a whole class of operators. A redeclaration of the operator for a user-specified type should not make the former completely invisible. We have chosen for the solution where the former

operator declaration is not hidden at all. Hiding of such an operator by a user-provided redeclaration of the operator is solved dynamically. This phenomenon is explained by an example. Let

```
function "<" (x, y : any_enum) return boolean;-- (1)
```

be one of the definitions for the "<" operator in the standard environment. Assume that a user defines his own enumeration type and redefines the "<" operator, as in:

```
type colour is (red, white, green, blue);

function "<" (x, y : colour) return boolean;   -- (2)
```

The fact that, with respect to the type colour, the latter definition hides the former one is *not* recorded during declaration processing. Consider the call

```
if red < white then ..
```

After application of the bottom-up algorithm and after reduction of the set of triples in the top-down scan *two* triples remain:

```
{ < "<", any_enum, colour> ,
  < "<", null, null> }
```

The "<" in the first triple refers to the definition for "<" labelled (1), the "<" in the second triple element refers to the definition for "<" labelled (2).

The top-down scan is extended to accommodate a solution for this kind of hiding. Prior to verifying that the set of resulting virtual definitions is indeed a singleton set, virtual definitions that are actually hidden are removed. The rules for hiding detection in a case like this are simple: a redeclaration always hides a declaration with formal universal parameter types.

The Ada language supports the notion of *derived types.* According to section 3.4 of [LRM-83] "*A derived type defines a new (base) type whose characteristics are derived from those of a parent type; the new type is called a derived type. ... Certain subprograms that are operations of the parent type are said to be derivable ...*".

For ordinary derived types the compiler explicitly redeclares the operators as triples. No particular problems with type checking therefore arise. For each application of an inherited function or operator this triple is used; the function or operator call is solved using this triple. This implies that in a resolved function call no reference remains to an inherited function or operator.

For numerical types the situation is different. Operators and subprograms on numeric types that are defined in package *standard* are not redeclared. The use of *formal universal types* in their declaration ensures that the type-checking mechanism is always able to relate the declaration to the newly-derived numerical type.

Baker's algorithm

Finally, we shall give a short comment on the algorithms proposed by Baker [Baker-82a], [Baker-82b]. Baker states that overload resolution can be done in a single bottom-up scan at compile time. No subsequent top-down scan is then required. The essential idea behind the algorithms is that the compiler front end constructs in a single left to right scan the set of all possible interpretations. Each interpretation is (conceptually) encoded as a (complete) tree. So, for each viable interpretation a private tree is constructed. Given a context type, the tree encoding the correct interpretation can be selected right away. This solution differs from other approaches where a single tree is used with attributed nodes to encompass the set of possible interpretations.

In Baker's algorithm the top-down propagation of information obtained from the context in which an expression appears is avoided. It therefore seems that a top-down scan through the tree can be avoided. However, what is additionally required is some form of garbage collection scan which definitely requires a scan through the set of discarded trees. Furthermore, a top-down scan through an expression tree is required to put the expression into some normalized form and to compute static and universal (sub)expressions.

Stockton [Stockton-85] claims to refine the above algorithm. To each node of the tree a list of interpretations is attached during the bottom-up scan. A clean-up scan is required, however, to propagate the remaining interpretations into the tree nodes.

4.5.2 Practical problems in overload resolution

A problem not set until now is how to handle some of the practical aspects of the implementation. Some practical problems are dealt with in this subsection.

A general problem, related to software engineering issues rather than to compiler writing, is how to handle complexity. The complexity in the implementation of overload resolution algorithms is high. In our implementation the overload resolution takes about 5,000 lines of C code. The complexity is caused by several factors, e.g. by:

- the inability to construct the final internal tree representation during the parse;
- selection of data structures for trees and sets;
- the high degree of interaction between the different parts of the implementation.

General structure of overload-resolution implementation

The set of functions and procedures that implements overload resolution forms a central part in the semantic analysis phase. During the walks to perform the overload resolution a number of other activities is performed as well. In, particular a large number of well-formedness checks is performed and the intermediate tree is put into some normalized form.

The parser hands complete expression trees to the semantic analyzer. Both the bottom-up walk and the top-down walk are done during recursive tree walks. The actions of the bottom-up scan are done on the postfix encounters during the first tree walk. The actions of the the top-down scan are done on the prefix encounters of the second tree walk.

A general naming convention is to call a function *bu_xxx* when it performs a bottom-up scan over a node labelled *xxx*. Similarly, a function *td_xxx* implements the top-down scan for the node labelled *xxx*.

The result of the bottom-up scan is a single tree, decorated with sets of types and sets of references to defining occurrences. The tree itself remains unchanged during this scan. Our choice, taking this approach rather than the pseudo-one-pass approach taken in the FSU compiler was based on the consideration that a top-down scan is required anyhow to perform some delayed checking and to normalize the internal tree structure.

The functions that implement the top-down scan can be seen as tree transformers. They take a tree as parameter and yield a possibly modified tree as result. Top-down processing is done on the pre-order encounters in the second recursive tree traversal. The post-order encounters are used for:

- determining the final form of the intermediate program tree representation;
- computation of static and universal expressions;
- putting discriminant constraints, aggregates and procedure calls in some normalized form;
- performing the remaining checks, still a large number.

Sets and set operations

The implementation of sets needs a special remark. We have chosen the straightforward implementation for sets, i.e. as linked lists. It turns out that the amount of time spent on set operations in medium-to-large sized programs is too large to ignore. Alternative implementations may turn out to be worthwhile. An implementation using some form of bit vectors seems, at first sight, an improvement. Two observations can be made however. The first observation is that the mapping between an element from the set domain and the particular bit in the bit vector is not known at compiler-generation time since the set domains are not known at compiler-generation time. The second observation is that set operations have to deal with *universal types*. For a set operation such as intersection, e.g. $S_1 \cap S_2$, the elements of both S_1 and S_2 have to be inspected for the occurrence of universal types. Consider the example

{any_integer} ∩ {real, short_integer, colour}

where *any_integer* is an actual universal type. The resulting set should contain *short_integer* as its element.

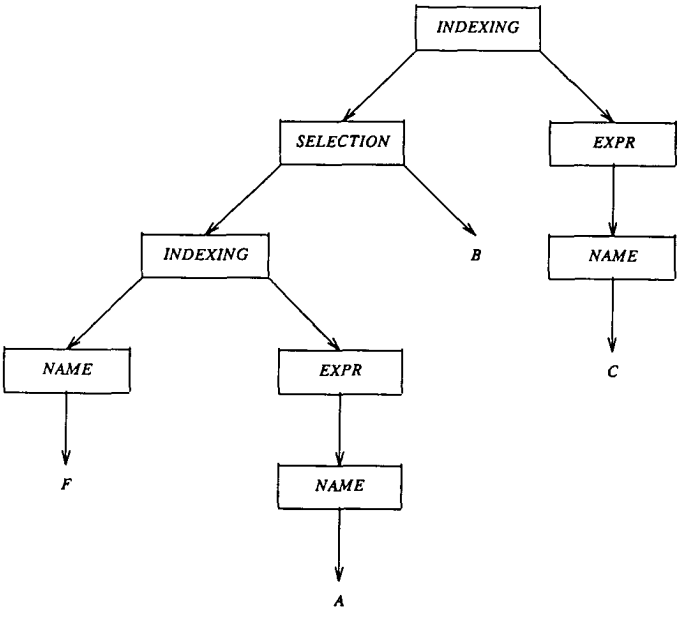
In our implementation we use two intersection operations, *propagate* and *reduce*. Set operations with formal universal types are performed by *propagate*. The transforming of actual universal types to specific types is done by the function *reduce*.

An example

By means of an example we shall illustrate the implementation of overload resolution in the Ada— compiler. Consider the statement

```
F (A). B (C);
```

From a purely syntactical point of view the construct stands for an indexing operation, followed by a selection, followed by yet another indexing operation. During syntactical analysis no attempt is made to distinguish between indexing, calling a function, or calling a task entry. The tree constructed by the parser, prior to overload resolution, is (schematically) given below. (Notice that the leaves of the tree are unresolved applied occurrences of identifiers.)



The construct is resolved in the following context:

```
...
task type X is
  entry B (Y : integer);
end;
type AX is access X;

type my_array is array (1 .. 10) of integer;
type my_rec is record
  B : my_array;
  ...
end record;
```

```
function F (X : integer) return AX;  -- referred to by F1
function F (X : integer) return my_rec;  -- referred to by F2

A: integer;
C: integer;
```

The name construct seems to have two possible interpretations. The first interpretation takes the first definition for *F*, calls the function, dereferences the result value and calls the indicated entry. In the second interpretation the second definition of *F* is taken, the function is called, in the result value the *B* component is selected and the indexing is performed in this component. Since the name construct appears on its own in a statement, the first interpretation is the correct one.

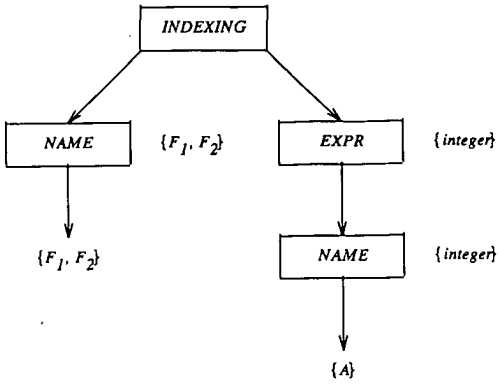
Overload resolution starts with calling *bu_indexing*. It then calls for *bu_selection* and *bu_expr* for its left and right subtree respectively. *bu_selection* calls for *bu_indexing*, *bu_indexing* calls for *bu_name* and *bu_expr* respectively. *bu_name* processes the set of visible defining occurrences of identifier *F*.

The actual bottom-up processing takes place on the postfix encounters:

- *bu_name* calls to look for the set of visible defining occurrences with identifier *F*. It gets the set $\{ F_1, F_2 \}$† and propagates the entities as result types to the *NAME* node. It must be realized that F_1 and F_2 do appear as type here although they are not legal as a user-defined Ada type;
- *bu_expr* recursively calls *bu_name* to resolve the primary consisting of the identifier *A*. It labels the *EXPR* node with the set of result types.

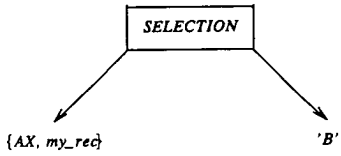
(In the diagrams sets are indicated by { .... }.) The tree fragment dealing with indexing is given below.

† The two occurrences of *F* are indicated by F_1 and F_2 respectively.

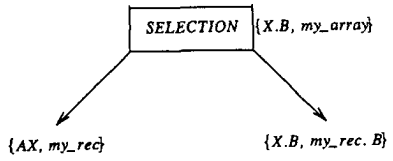


bu_indexing tries to reduce the set of entities to which an indexing (or calling) operation can be applied, by discarding all entities for which the result types of the argument (parameter) expression do not contain the corresponding index (parameter) type. In this particular case F_1 and F_2 accept *integer* as argument (parameter) type. The set of result types of the indexing (calling) operation is {*AX, my_rec*}.

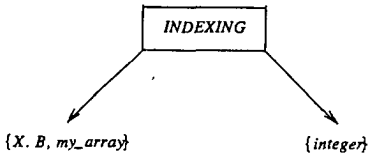
The context in which the selection operation has to be performed is schematically:



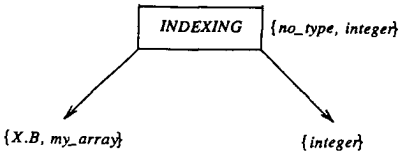
For each of the types of the left operand it is checked that the type allows selection as operation. If required, a single dereferencing is performed (section 4.1 of [LRM-83]). Next it is checked that in each of the remaining left-operand types a component *B* can be selected. Both left-operand types in this example allow selection of a component *B*. The literal *B* is now replaced by the set of selected items and a set of result types is determined. Again here, the selected entity *X. B* is treated as a type, resulting from a selection operation.



bu_indexing is then applied to the *INDEXING* top node. First *bu_expr* is called to perform a bottom-up walk over the expression *C*. The structure on which *bu_indexing* is operating is then (schematically):



bu_indexing checks for each of the left-operand types whether an indexing or a calling operation is allowed or not. In this particular example *X.B* allows calling, *my_array* allows indexing. It is checked that the calling and indexing operations are applicable with respect to the number of and the types of the right-operand expressions. Both left-operand types are indeed applicable. The result type of an indexing operation is *integer*, the element type of the array type *my_array*. The result type of applying an entry to its parameters is by definition *notype*, a compiler-internal type.



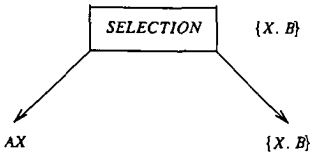
The context in which the whole construct appears is now to be taken into account. Since the construct appears as a statement, the result types of the construct are intersected with

{*no_type*}, leaving {*no_type*}.

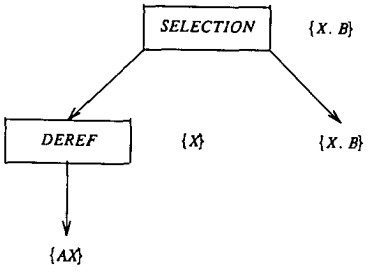
A top-down scan over the whole tree is performed to propagate the context type and, most importantly, to re-shape the tree.

First, *td_indexing* is called. For each element of the left-operand type set { *X. B*, *my_array* } it is inspected whether the type *no_type* is obtained after indexing (or calling) or not.

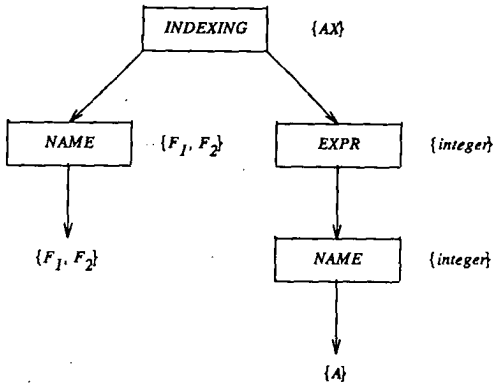
The resulting "type" *X. B* is then used to reduce the left subtree and *td_selection* is called for this subtree. Given the result type of the operation (*X. B*), the selected item is uniquely identified and the result type of the left subtree is determined:



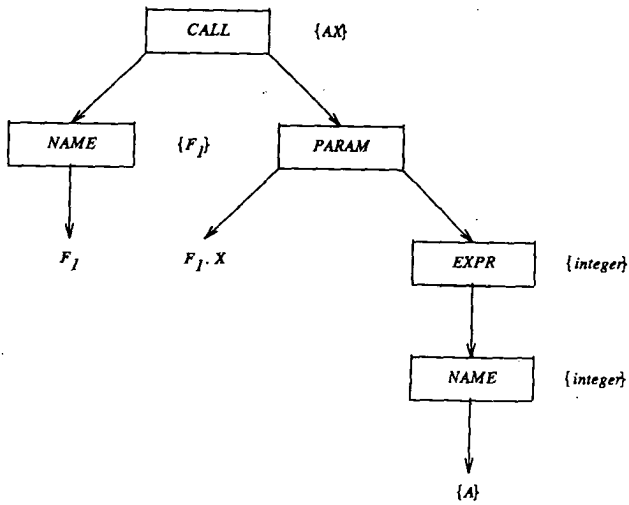
It is detected that prior to performing the selection a dereferencing has to be performed; the dereferencing is made explicit in the tree.



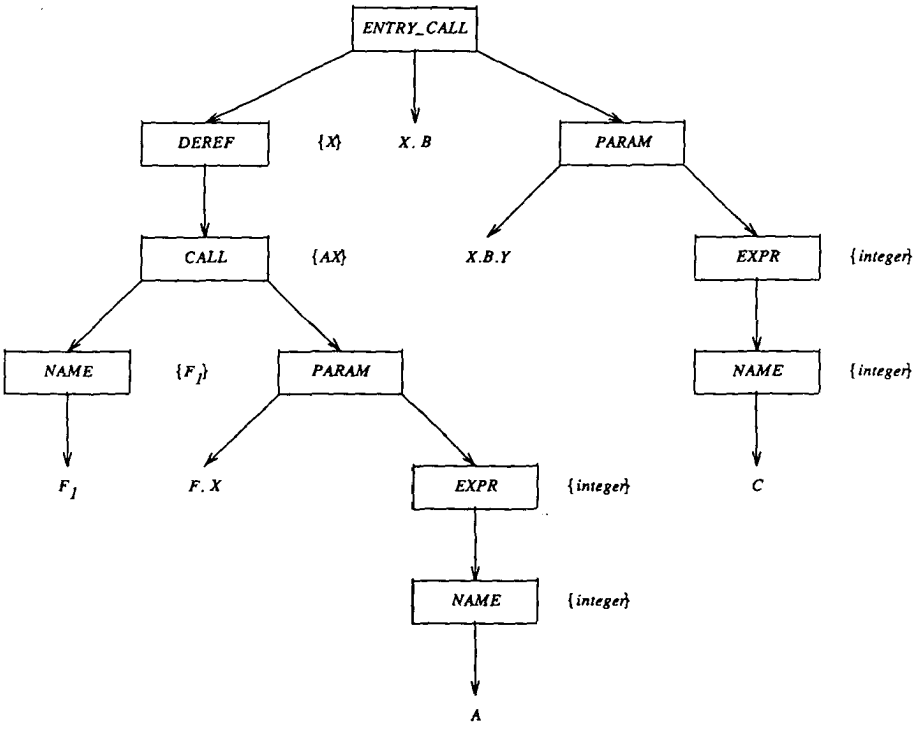
The result type *AX* is used to reduce the *INDEXING* subtree.



Furthermore, it is detected that this particular result type is obtained by applying F_1 to the given argument and the tree is locally restructured. The fact that it is a function call is reflected by a *CALL* node. Furthermore, the parameter binding is made explicit in the tree structure.



A final shaping of the top nodes of the tree takes place on return from the top-down scan. The resulting tree is given below.



Normalization of the intermediate tree representation

The compiler front end acquires some knowledge on the constructs dealt with. In order to simplify further processing, the final bottom-up tree walk (following the top-down scan required for overload resolution) normalizes the tree representation of some constructs. The normalization of tree constructs implies the reorganization of these tree constructs into some canonical form. The prime motivation for putting constructs into a canonical form is to ease further processing of the tree, e.g. by a code generator.

In particular aggregates, function calls and discriminant constraints are normalized. The normalization of an aggregate implies:

- for a record aggregate making a component association for each record component. (Note: if a record type contains variants, then the aggregate of the type has constants

for the discriminant values on which selection is based). Consider the example of a record type definition

```
type my_rec is record
  a, b, c, d : integer;
end record;
```

The aggregate appearing in the declaration

```
V : my_rec := (a | c => 1, others => 2);
```

is transformed into

```
V : my_rec := (a => 1, b => 2, c => 1, d => 2);
```

The **others** choice is replaced by associations for the corresponding record components.

- for an array aggregate of the form

```
(M .. N => e)
```

the alternative form

```
T (M .. N)' (others => e)
```

is constructed (although the transformed form is formally not a legal construct in the Ada language.) This form is similar to the form of other array aggregates.

Normalization of the intermediate representation of procedure and function calls implies the ordering of parameter bindings and the supply of those actuals that are defaulted. Consider the example

```
procedure p (a, b : integer := 3; c : colour := red; d : integer := 4);
```

The call with default parameters

```
p (d => 7, b => 3)
```

is expanded into a call with explicit parameters in which the parameters are ordered.

```
p (a => 3, b => 3, c => red, d => 7);
```

Finally, normalizing discriminant constraints is similar to normalizing record aggregates.

4.5.3 The implementation of scope and visibility

Most programming languages of the last decade have some form of explicit scope control in addition to automatic scope control provided by classical block-structured languages. The Ada programming language, a language designed in the last decade, is no exception to this. It supports both explicit and implicit scope control.

The implementation of automatic scope control in programming languages is well understood and widely known. On the other hand, implementation schemes for explicit scope control are less wide spread. In the literature only a few entries could be traced that contained a description.

A good characterization of the complexity of symbol table structures for a language as complex as Ada is given by Baker in its description of the FSU Ada compiler [Baker-82a]: " *The data structures used to represent semantic information, which might be termed 'the symbol table' in a broad sense, were very complex and space consuming*".

Already in 1977 Wirth complained about the complexity of the symbol table organization for the relatively simple Modula compiler. To quote Wirth [Wirth-77]: "*The additional sophistication of the compiler's symbol table mechanism is rather considerable, and in any case greater than anticipated.*".

In early versions of the DAS compiler visibility was implemented using a linear-search-based scheme. For each applied occurrence of an identifier a search was performed through the relevant parts of the intermediate tree representation. Although our expectations of the efficiency were not high, it was felt to be surprising that over 25 % of the front end CPU time was used for declaration processing and identifier look-up. Measurements on several 1,500 line programs showed that just the look-up of all identifiers required about 250,000 times a comparison between an applied and a defining occurrence of an identifier. The development of a more efficient scheme seemed certainly worthwhile. Therefore, a more efficient scheme was designed and implemented. Even a simple prototype implementation of this scheme gave a performance improvement of the compiler front end of over 20%.

We present two schemes for the implementation of scope and visibility. The first one is a simple linear-search based scheme. The second one is based on the use of a single global hash table to provide fast access to all visible declarations with a given identifier.

Due to the lack of published material, it is not possible to give a reasonable comparison of our results with the results of others. Therefore, this subsection concentrates on discussing the techniques that were implemented in our compiler. The structure of this subsection is as follows: first the problem is stated and a survey of the literature available is given; then attention is given to algorithms for implementing the scope and the visibility rules and finally, the problem is reconsidered in the context of very large programs.

Problems and a survey of the literature

As stated before, it is amazing that the subject of modelling name tables for language with explicit scope control has got so little attention in literature. Modern compiler writers' textbooks, [Goos-83] and [Aho-85], hardly touch the subject. The only entries found which discuss name table management for languages with explicit scope control, were papers by Wirth [Wirth-77] and Graham et al [Graham-79a]. Some papers were traced that discussed implementation of scope and visibility for the Ada language, papers by Baker [Baker-82a] and Blower [Blower-84] are examples.

The precise scope and visibility rules of the Ada language are given in chapter 8 of [LRM-83]; the reader is supposed to be more or less familiar with these. Roughly stated, the rules differ from the rules in e.g. Pascal in the following ways:

1. due to the module structure of programs in the language, the scope of a declaration is not necessarily restricted to the textual region in which the declaration occurs.

2. for each identifier and each place in the text, the visibility rules determine a *set* of declarations rather than a single one with the given identifier that defines possible meanings of an occurrence of the identifier (section 8.3 2 [LRM-83]).

The *immediate scope* of a declaration is defined in section 8.2 of [LRM-83]. It starts with the declaration and it extends at least to the end of the declarative region in which this declaration occurs. A declarative region is a portion of the program text. The definition (section 8.1 of [LRM-83]) states i.a. that a package declaration together with its body forms a single declarative region. From an implementor's point of view this implies that the classical implementation of looking for the defining occurrence of an identifier by simply searching inside out through nested scopes is not sufficient any more.

A complication is that the search for defining occurrences with a given identifier does not have to stop when the first one is found. The declaration that is found may allow overloading in which case the search for other declarations with the same identifier has to be continued.

Yet another point to discuss is the finding of declarations that are made *directly visible*† through *use clauses*. A declaration that is made directly visible through a *use clause* will be called an *imported declaration*. According to section 8.4 of [LRM-83] a declaration in the visible part of a package whose name appears in a use clause is made directly visible unless:

- the place considered is within the immediate scope of a homograph‡ of the declaration.
- there exists a declaration that has the same identifier in the visible part of another package that is being mentioned in a use clause, and not both declarations are declarations of enumeration literals or subprograms.

The first constraint ensures that an imported declaration never hides a declaration in its immediate scope. The second constraint ensures that two declarations with the same identifier that are potentially visible through the application of a use clause, hide each other *unless* each of them is the declaration of an enumeration literal or subprogram. (An implication of this constraint is that using a package at a given point in a program text can reduce the set of directly visible declarations rather than increase that set. Declarations made visible through a use clause may even get invisible as a declaration is added to one of the packages being *used*.)

In figure 15 a program fragment is given in which the second constraint is illustrated.

---

† directly visible is a term defined in [LRM-83]. Visibility is direct if no selection is required to make the intended declaration of that identifier visible.

‡ Each of two declarations is said to be a homograph of the other if both declarations have the same identifier and overloading is allowed for at most one of two. If overloading is allowed for both declarations, then each of the two is a homograph of the other if they have the same identifier, operator symbol or character literal as well as the same parameter and result type profile.

```
procedure R is
  package traffic is
    type colour is (white, yellow, green, brown, black);
    ...
  end TRAFFIC;

  package water_colours is
    type colour is (white, red, yellow, green, blue, brown, black);
    ...
  end water_colours;

  use traffic;       -- colour, red, amber and green are directly visible
  use water_colours; -- two homographs of green are directly visible
                     -- but colour is no longer directly visible

  subtype light  is traffic. colour;
  subtype shade  is water_colours. colour;
  signal :        light;
  paint :         shade;

begin
  signal := green;
  paint := green;
end;
```

Figure 15. *Example of importing declarations*

A number of checks has to be performed during processing of a declaration. In general, it should be verified that the declaration does not violate the context conditions of the language. To give an example, two declarations that are homographs of each other cannot be legally declared in the same declarative region.

Processing the declaration of a derived type becomes somewhat more complex. It may be necessary to declare implicitly a large amount of enumeration literals, operators, and subprograms. The well-formedness rules for implicitly declared operators, functions, procedures and enumeration literals differ from rules for normally declared entities. An implicitly declared entity may be hidden by an explicitly declared entity in the same declarative part. The converse is never true. Therefore, in declaration processing a distinction has to be made between processing explicit and processing implicit declarations. In figure 16 an example is given of an implicitly declared function.

```
type colour is (red, white, blue);
function f (x : colour) return colour;  -- an operation on colour

 .....
package derive_colour is
  type my_colour is new colour;  -- derived from colour,

-- implicitly declared
-- function f (x : my_colour) return my_colour;
--
  a, b : my_colour;
end derive_colour;
use derive_colour;
 ....

 begin
  a := f (b);    -- this must be a call to the implicitly
                 -- declared function f
end;
```

Figure 16. *Derived types: an example*

In a sense processing a declaration and looking for declarations with a given identifier are complementary actions. The form in which a declaration is stored in a particular implementation depends on the way defining occurrences of identifiers are searched for.

Some ideas on hashing in symbol tables for languages with explicit scope control are given by Graham et al. [Graham-79a]. The techniques presented there are, however, oriented towards Modula I. Nevertheless, they form the basis of the model discussed in this subsection.

Recent research is directed towards a more formal approach in the construction of name lists. From a formal point of view a name list can be seen as an instance of an abstract data type. A paper discussing automatic generation of a name list, based on a more or less formal specification is from Reiss [Reiss-83].

The problem of managing name spaces in very large programs has recently also got some attention in literature. Without further discussing the subject in any depth, we refer to papers by Kamel [Kamel-84] and Conradi [Conradi-85].

The BreadBoard compiler [BreadBoard-82] is characteristic with respect to the implementation of scope and visibility for a whole class of compilers. The name list is not viewed as a single data structure. Quinn [Quinn-82] states: "*The symbol table plays a major role in semantic checking: it is the most important data structure used by the checker ..... Pass 2*[†] *does not have a single, monolithic symbol table. Instead the checker creates and uses*

† Actually, the third pass, doing semantic analysis is often referred to as Pass 2. Between Pass1 and this Pass 2 a small intermediate pass operates.

many symbol table instances as it analyzes the input program. There are two reasons for this: First a symbol table can describe only part of the name context, such as the declarations in one declarative region.... Second, when a symbol table is modified, the result is usually a new symbol table, not an updated old one....". The approach taken by the designers of the BreadBoard compiler differs from our approach. We believe that a *single* table accelerates identifier look-up, while the cost for keeping the structure up to date remains acceptable. The situation may be different though when dealing with very large programs, programs with thousands of library modules.

An approach, similar to the one taken in the BreadBoard compiler, is taken in the DDC compiler [Bundgaard-82] and the AIE compiler [AIE-82], [Blower-84]. In these compilers the mechanism for identifier look-up is extended with a caching mechanism. According to Intermetrics [AIE-82]: "*...The actual implementation of look-up involves caching answers from previous look-ups and caching look-ups for USE visibility. The details of caching are unimportant at this level*". The paper by Blower [Blower-84] is completely devoted to a discussion of the implementation of efficient identifier look-up in the AIE compiler.

The approach taken in the York compiler differs from both the approaches taken in the BreadBoard, the DDC and the AIE compiler, and from the approach taken in our implementation. Declaration processing and identifier look-up in the York compiler is in two separate passes. According to Murdie [Murdie-83] a number of problems occurs in maintaining a correct set of visible identifiers at any point in the program.

The description of the name-list structures in the one-pass front end described by Baker [Baker-82a] is short: "*...For each symbol in the table there is a list of all its current potentially visible definitions. This list is kept up to date continually by adding and deleting local definitions at one end and definitions made visible by use clauses at the other.*" It is computed dynamically whether a declaration is visible or not.

The name-list structure in the Ada+ front end [Barbacci-85] is also intertwined with the intermediate tree representation. Table look-up is speeded up by hashing on the basis of a single declarative part at the time. Whether two declarations are homographs of each other or not and, therefore, one of them hides the other, is dynamically computed.

Implementation strategies

The bindings between the applied occurrences and the corresponding defining occurrences are maintained in the intermediate program representations. They are encoded as semantic links in the various nodes; once the applied occurrences of the identifiers in a program are bound to their corresponding defining occurrences, the need for an explicit name list is gone. What remains is the intermediate program representation, containing all required information.

In the Ada(em compiler a name list is maintained during front-end processing to speed up the process of binding an applied occurrence of an identifier to the set of corresponding defining occurrences of that identifier. The functionality which is required for a name list manager is simple; it must support:

1. look-up for all defining occurrences of a given identifier which are visible according to the scope rules, either in their immediate scopes or made directly visible through a *use clause*;
2. maintainance of the consistency of the data structures involved on changes in the scopes; i.e. on compiling† the start and the end of a construction defining a scope (scope entry and scope exit for short).

In this subsection we discuss two implementation strategies for the implementation of visibility. First we define visibility in more formal terms, then we give a first implementation strategy based on a linear search, and next we present a more efficient strategy.

Let

$S_1 = \{x_1, x_2, \ldots, x_n\}$

be the set of declarations whose immediate scope contains the current point in the program. (The assumption is that if $i < j$ then the declaration x_i was elaborated before the elaboration of x_j)

Let

$S_2 = \{y \mid y$ *is a declaration in the visible part of a package mentioned in a use clause* $\}$

be the set of declarations in the visible part of the packages that are named in *use clauses.*

DVI (t), the declarations with identifier t that are directly visible at the current point in the program can be defined as:

$DVI(t) = VII(t, S_1, S_2) \cup VI(t, S_1, S_2)$

where *VII* (*Visible In Immediate scope*) is defined as:

$$VII(t, S_1, S_2) = \{x_i \mid x_i \in S_1 \text{ and } identifier(x_i) = t \cap \text{not } (\exists\ j \text{ such that } j > i \text{ and } is\_homograph(x_i, x_j))\}$$

and where VI (*Visible through Import*) is defined as:

$$VI(t, S_1, S_2) = \{x \mid x \in S_2 \text{ and } identifier(x) = t \text{ and not } (\exists\ y \in NIV(t, S_1, S_2) \text{ such that } is\_homograph(y, x)) \text{ and not } (\exists\ y \in S_2 \text{ for which } hides(y, x))\}$$

In these definitions:

is_homograph is defined as

† In this subsection we mean by *compiling* the actions to be taken during semantical analysis.

is_homograph (x, y) is **true** *iff x and y are homographs of each other;*

hides is defined as

hides (x, y) is **true** *iff x and y are not both subprograms or enumeration literals.*

Furthermore,

identifier is a function yielding the identifier of the declaration that is given as argument.

Updating the sets S_1 and S_2 is as follows:

- on the elaboration of a declaration the declaration is added to S_1;
- on the elaboration of a *use clause*, the declarations in the visible part of the packages that are named in the use clause are added to S_2;
- on scope exit:
  1. all declarations that were elaborated in the scope are removed from S_1;
  2. all declarations belonging to packages named in *use clauses* that were added to S_2 in this scope are removed again.

A first implementation model

A first implementation is directly based on the formal description above. Two steps are taken:

1. in the first step the set of declarations is collected with a given identifier whose immediate scope contains the current point in the program, i.e. *VII* is computed;
2. in the second step *VI* is computed, the set of defining occurrences of the given identifier that is made directly visible through *use clauses.*

The finding of declarations in their immediate scopes

To each (part of a) declarative region a *mapping* from identifiers to their corresponding defining occurrence is associated. (Such a mapping may be implemented as a hash table.) These mappings are used to speed up the finding of declarations with a given identifier. As in the definitions above, once a declaration is found it has to be verified whether it is in fact visible or not.

For the management of the mappings of those scopes that are being compiled, i.e. the *open scopes*, a *stack of mappings* (the (*scope stack*†) is maintained. Finding the defining occurrences with a given identifier is then as follows: the mappings that are on the stack are successively applied (from top to bottom) to the given identifier. When a declaration with the given identifier is found, it is dynamically computed whether the declaration is a homograph of any declaration found earlier or not. If it is, the declaration remains hidden, otherwise it is added to the set of found and visible declarations.

This approach approach was made in early versions of the DAS compiler, in the AIE compiler, in the DDC compiler and probably in many more. Identifier look-up is rather

† In the figures the stacks grows downwards.

expensive in time; both the AIE compiler and the DDC compiler use some caching scheme to improve the performance. A significant cost turns out to be the dynamic recomputation of whether a declaration is visible or not.

On scope entry, a mapping for the newly-opened scope is pushed on the scope stack. If the entered scope is a *body*, i.e. a package body, a procedure body or a task body, the mapping(s) for the corresponding specification are pushed first. For example, on entry of the scope of a package body, the mappings associated with the package specification and the private part of the package specification are pushed first. On entry of a subprogram body, the mapping of the corresponding specification is pushed first.

Processing a declaration causes the identifier of that declaration to be added to the mapping of the scope of the textual region currently being compiled.

On exit from a scope-defining construction, the scope stack is updated by removing all mappings related to the construct about to be left. The stack of open scopes at the line marked *1* in the program fragment below is given in figure 17.

```
package p1 is
  p0, q0, r0 : boolean;
end;

package body p1 is
  p1, q1, r1 : integer;
  ---     (1)
end ;
```

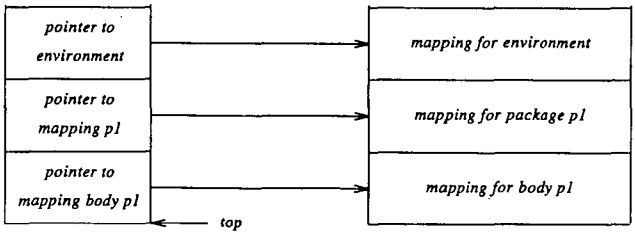


Figure 17. *Scope stack*

The finding of imported declarations

A first implementation of look-up for imported declarations is also based on the formal specification above. A set of potentially visible declarations is determined and in a second step the hidden elements are removed. Again, the mappings between identifiers and their corresponding declarations (introduced in the previous subsection) are used. When a use

clause is being compiled, the mappings for the packages whose names are mentioned in the use clause are pushed on a stack, the stack of *used packages*. On scope exit from a construct in whose scope the *use clause* was processed, the table entry for any *used* package that was pushed on the stack in the scope to be left, is popped from this stack.

To find a set of defining occurrences of a given identifier, the mappings on the stack are successively applied to the given identifier. When the whole set has been found, checks are done to investigate whether declarations are really visible or not. Due to the second constraint given in section 8.4 of [LRM-83] (*...there exists a declaration with the same identifier in the visible part of another package that is being mentioned in a use clause, and not both declarations are declarations of enumeration literals or subprograms*), this computation is delayed until the whole set has been determined.

An improved algorithm

It was hardly surprising to find that a substantial amount of time was spent in declaration processing and identifier look-up. In particular the *on-the-fly* computation of whether a declaration is indeed visible or not is expensive in time.

A second set of algorithms was devised and implemented, taking the bottle necks of the previous approach into account. The algorithms are based on the observation that it is possible to maintain a single set S that contains only the visible declarations. The price to be paid is that of keeping this set consistent with the set of actually visible declarations at any point in the program.

The implementation uses a single hash-table which partitions the set of visible declarations into equivalence classes. The whole set of *visible* declarations of a given identifier can then be found in a single scan through the equivalence class the identifier belongs to.

Structure of the implementation

Three tables are maintained in the implementation. A *hash table* maps a given identifier into an ordered list of declarations of which the identifiers of the declared entities hash to the same hash value. A *name list* contains descriptions of all identifiers in their potential scope. (The *potential scope* of a declaration corresponds to the construct determining the extent of the declared entity. A declaration should remain in the name-list structure as long as its potential scope is in operation, since in general the declaration can be made directly visible again during that time.). Finally, a *block table* contains descriptions for all scopes that contain potentially visible declarations.

Our implementation uses a single mapping which is implemented through a hash table to compute the set of visible declarations of a given identifier. Hashing itself, although important in its own right, is not discussed here. The references contain entries for so-called perfect hash functions for keywords in the Ada language [Wolverton-84], [Sebesta-85].

For each scope in the block table, the description contains a pointer to an element of the name list. This pointer is used to locate the declarations belonging to that scope.

Each non-null entry in the hash table contains a link to a name-list element whose identifier hashes to the index of the hash table entry. A name-list element contains 5 fields:

1. a *hash chain* element, a link to the next element in the name list which hashes to the same hash value;
2. an encoding of the scope in which the entity is declared;
3. the *identifier* and other attributes of the entity itself;
4. a *flag field*, containing flags indicating the status of the entity;
5. a *hide field*, containing the number of declarations hiding this declaration.

The *hash chains* are used to implement an ordering in those declarations that hash to the same value.

The encoding of the scope in a name-list entry to which the corresponding declaration belongs is for practical reasons only. Some operations require knowledge of the containing scope of a declaration once it is found.

The *flag field* contains flags indicating the status of the name-list entity. Possible values are *open_scopes, imported, asleep* and *unlinked.*

The *hide field* contains an indication of whether a declaration is visible or not. The most obvious approach to the implementation of hiding is to implement a binary relation between the hiding declaration and the hidden declaration. Unfortunately, a single declaration may hide several others; therefore such a relation is expensive to implement. Implementing the transposed relation

X is_hidden_by Y

is cheaper. Each name-list entry *X* gets in its *hide field* a number indicating the number of elements *Y* such that

$X\ is\_hidden\_by^{+}\ Y$

holds†. A symbol table entry with a value N in its hide field (N /= 0) is always hidden by exactly one declaration with the value N - 1 in its hide field. Any declaration with N (N > 0) in its hide field is kept hidden. Hiding between imported declarations is somewhat more complex; its discussion is postponed until the processing of *use clauses* is discussed (page 98).

Consider for example the program fragment of figure 18. The declarations on lines *d* and *b* are homographs; the declaration on line *d* consequently *hides* the one on line *b.*

† As usual "+" denotes the transitive closure of the relation.

```
L: declare
   function f1 return integer;      -- a
   function f1 return boolean;      -- b
   type y is (f1, f2);              -- c
   ...                              -- (1)

   begin
      ...
M:    declare
      function f1 return boolean;                -- d
      ...
      begin
      ...
N:       declare
         function f1 return integer;             -- e
                                  -- (2)

      ...
```

Figure 18. *An example of hiding declarations*

Textually preceding the block labelled *N*, the *f1* declared at the line marked *a* is visible and the value in its hide field is 0. The value in the hide field of the element in front of it in the hash chain, the element describing *f1* on the line marked *b*, contains 1 since there is at this point one symbol table element with a hiding declaration. On the elaboration of the declaration at the line marked *e* the *f1* on line *a* will be hidden; the value in its hide field will increase by one.

The encoding scheme for hiding a declaration must take into account that a hidden declaration will become visible again after the scope of the hiding declaration is left. For instance, after leaving the scope *N*, the declaration of *f1* at line *a* is visible again.

The name list representing the state at the line marked 2 is as given in figure 19. The declarations *f1* and *f1* on the lines marked *a* and *b* respectively are hidden by the declarations of the same identifier on the lines marked *e* and *d* respectively.

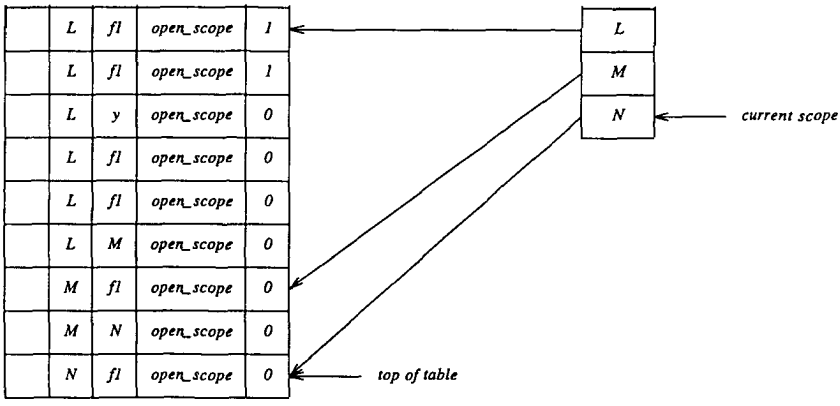


Figure 19. *Encoding visibility information in a set of declarations*

The value *open_scope* in the flag field indicates that the immediate scope of the declaration is in operation. It can be seen that the function *fl* declared at line *a* and the function *fl* declared at line *b* are hidden, each by one other entity.

The price for an efficient identifier look-up is paid by maintaining the consistency of the name-list structures, in particular on entry and exit of a construction defining a scope and at processing a declaration.

Scope entry, scope exit, adding an identifier

In this subsection we look at the problem of updating the name-list structures on entry and exit of a construction defining a scope and of adding a declaration to the name-list structures. Throughout this subsection, however, only immediate scopes are taken into account.

On the entry of a construction defining a scope, e.g. the entry into a block, actions to be taken are similar to the actions to be taken in a symbol table for e.g. Pascal. A new entry is pushed onto the scope stack and it is initialized to describe the newly opened scope.

On entering a declaration, it has to be verified for *each* declared entity with the same identifier (found through the hash chain) whether its hide field value has to be increased or not. In a more algorithmic style:

process a new declaration X as follows:

forall *elements Y in hash chain with identifier X*
loop

```
  if Y.flag /= open_scope
  then
    if is_homograph (X, Y)
    then
      Y.hide_field = Y.hide_field + 1;
    end if;
  elsif ... -- take hiding of imported
          -- declarations into account
  end if;
end loop;

add X to the name list and
enter X into the hash chain
```

The scope of a declaration that appears in a specification extends to the corresponding body. On entry of the scope of a body construction, the declarations of the corresponding specification must be made visible again. On the elaboration of the exit from a specification (e.g. from a package), the declarations of the construction are *not* removed from the name list; they are put *asleep* by unlinking them from the hash chain and marking them *asleep*. Furthermore, the scope description itself on the scope stack is also marked *asleep*. In a more algorithmic style:

```
put a declaration X asleep :

  unchain X from the hash chain.

  forall elements Y in hash chain with identifier X
  loop
    if Y.flag = open_scope
    then
      if is_homograph (X, Y)
      then
        Y.hide_field = Y.hide_field - 1;
      end if;
    elsif .... -- take into account imported declarations
    end if;
  end loop;

  X.flags = asleep;
```

The gain of this approach is a simple *back up* to situations where the declarations are made visible again.

A side effect of marking a scope description and marking the declarations of a scope rather than removing them is that the current scope, i.e. the scope description reflecting

the current position in the source text, is not necessarily the top-most element of the scope stack. Consider the program fragment of figure 20. The symbol table representing the visibility at the line marked *a* is given in figure 21.

```
procedure main is
  package pp is
    p1, p1, r1:  T;
  end;

  p1, q1, r1:  T1;

--              (a)
  package body pp is
    p2, q2, r2 :  T3;

--              (b)
  end;

  use pp;
--              (c)
  ...
```

Figure 20. *A program fragment*

The declarations *p1*, *q1*, and *r1*, all from the scope labelled *pp*, are on the name list marked *asleep*. The top most element on the scope stack is the description of the scope labelled *pp*; the current scope is, however, *main*, the second element on the scope stack.

On exit from the scope of a body construction, the name list is cleaned up. Since body constructions act as the potential scopes for the declarations in enclosed specifications, all declarations local to these constructions can then be removed from the name list.

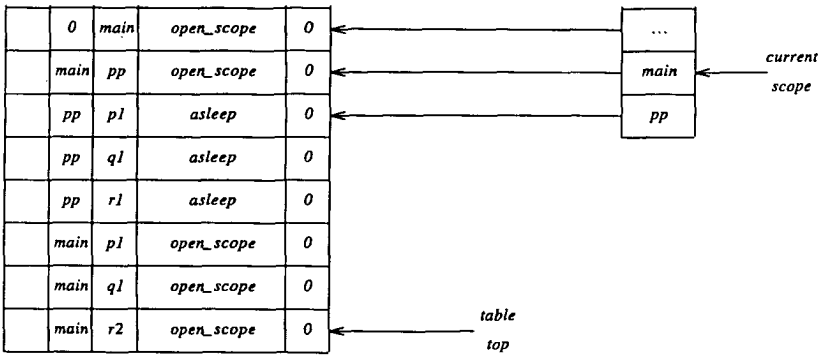


Figure 21. *Symbol table at position (a) of program from figure 20*

On the elaboration of the entry into a package body, e.g. the body of *pp* (figure 20), the declarations of the corresponding specification have to be made directly visible again. This is done by re-entering the declarations into the hash-chains. Such a *wake-up* action is done in a single scan through the name list, just as the corresponding *asleep* action. During this scan a dynamic identification of the scope to which an entity on a name list belongs is required; a package may contain other packages; if so, the declarations of the latter must remain *asleep*.

Consider again the program fragment of figure 20. The name list representing the visibility at the line marked *b* is given in figure 22. The ordering of the symbol table entries through the hash chain is important; *pp. p1* hides *main. p1* although the declaration of the former appears textually later than the declaration of the latter. Therefore, *pp.p1* should appear ahead of *main. p1* in the hash chain.

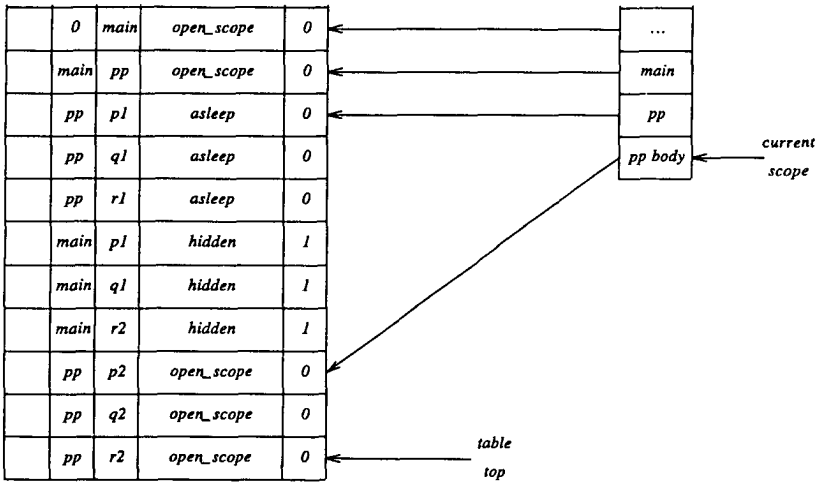


Figure 22. *Name list at line b of program fragment figure 20.*

The processing of a use clause

Declarations in the visible part of a package can be made directly visible again by mentioning the package name in a use clause. The handling of imported declarations complicates the single mapping scheme, primarily because of the different hiding rules that apply to imported declarations. First of all, a difference has to be made between declarations that must be imported and that are already on the name list and declarations that must be imported and are not on the name list.

In the first case only declarations that are marked *asleep* are interesting. Declarations that are already marked *open_scope* or *imported* can be skipped. If the containing scope was in operation, the *use clause* can be ignored. If the containing scope was a scope corresponding to a package already being used, the same applies.

The second case, i.e. the declarations to be imported are not yet anywhere in the name list, occurs when a package, that is named in the context specification of the compilation unit, is mentioned in a *use clause*. The declarations of the visible part of such a package are put on the name list.

The process of importing declarations (under the assumption that they *are* already on the name list) is given below:

forall ***declarations X to be imported***
loop
  forall ***elements Y in the hash chain***

```
        such that identifier (X) = identifier (Y)
    loop
      add X to hash chain
      if Y.flags = open_scope and then is_homograph (X, Y)
      then
        X.flags = imported;
        X.hide_field = Y.hide_field + 1; -- not visible
      elsif Y.flags = imported and then X hides Y
      then
        -- X and Y hide each other
        Y.hide_field = Y.hide_field + 1;
        X.hide_field = Y.hide_field; -- symmetric in hiding
      end if;
    end loop;
  end loop;
```

The value of the hide field contains the number of declarations hidden by a particular declaration. Notice that, with respect to imported declarations, the hiding is symmetric. As a consequence the values in the hide field of two imported declarations that hide each other are the same.

The algorithm for adding a declaration that hides already existing imported declarations does not change. On adding a declaration *X*, the value of the *hide field* of any declaration *Y* for which

```
is_homograph (X, Y)
```

holds is increased.

It is easy to see and to prove that under all circumstances the value of

```
X.hide_field
```

is exactly the number of declarations which keep *X* hidden. A schematical representation of the name list at the line marked *c* of the program fragment of figure 20 is given in figure 23. Notice that some of the declarations of the visible part of package *pp* remain *hidden.*

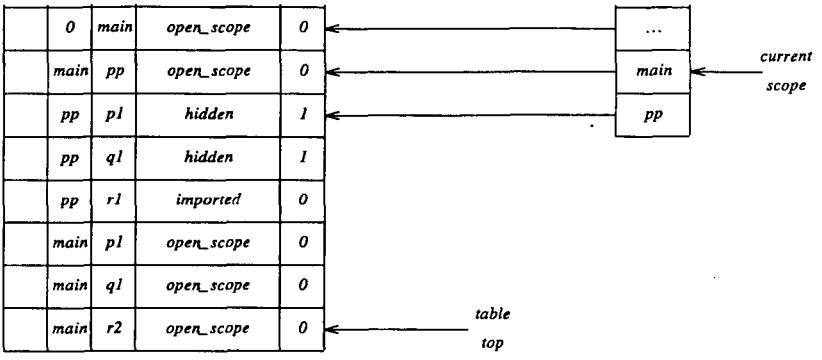


Figure 23. *Name list at line c of program fragment figure 20*

Scope exit revisited: an example

On the end of a construction which defines a scope in which *use clauses* were elaborated, the declarations that were imported by the elaboration of the *use clauses* have to be removed from the hash chains. Consider again the program fragment of figure 20 at the end of the procedure *main*. The *current scope* i.e. the scope to be left, is *main*, the scope topmost on the scope table is *pp*. It is, however, easy to see that any scope appearing above the one to be deleted is a scope belonging to some enclosed specification.

Processing the name-list mechanism at exit of a construction that defines a scope can be formulated as:

```
forall scopes X above the description of scope P on the scope stack
loop
  forall elements Y on the symbol table in scope X
  loop
    if Y.flags = imported or else Y.flags = open_scope
    remove Y from the hash chain
    Y.flags := unlinked
  end loop;
end loop;
```

The searching for declarations in a specified scope

Searching for an identifier within a specified scope is, as such, hardly worth mentioning. However, given our implementation strategy, *three* cases have to be distinguished:

1. the scope in which the declarations for the given identifier is looked-up is an open one, the declarations found on the hash chain for a given identifier contain as a subset the declarations from the specified scope.

2. the scope in which the declarations for the given identifier are looked-up is marked *asleep*, the declarations can be found on the symbol table. The elements cannot be found on the hash chains, though. A linear search seems inevitable.
3. the scope in which the declarations for the given identifier are looked-up has never, so far, been on the symbol table. The declarations are searched for by a linear search through the tree fragment encoding the scope.

...

```
Q: declare
     type X is range 1 .. 10;
     A : X;
   begin
     A := Q."+" (1, 2);  -- legal
   end;
```

Figure 24. *Complex case of a qualified name*

A complication arises when searching for operators on numerical types (figure 24). It was stated before that no redeclarations of numerical operators are entered in the name list; overload resolution can deal with operators inherited for numeric types. This implies that, when looking for an element from a given set of numeric operators, a search algorithm has to be applied that is capable of returning virtual definitions even when they are not actually there. This problem has been identified by Schonberg et al [Schonberg-82]; our solution is based on this work. Finding the declarations for a numerical operator in a specified scope according to this solution takes place in two steps:

- collect the set of declarations for that operator in the specified declarative part;
- collect in a second step the set of declarations of numerical types in the specified scope and create for each type found a *virtual* definition.

Note that the overload resolution algorithm itself takes care of the hiding of imported declarations.

Efficiency of identifier look-up

In this section two methods were presented to implement identifier look-up for the Ada language. The second method was an order of magnitude faster than the first one. Essential for the efficiency of the second algorithm is that the efficiency gain in identifier look-up outweights the increased cost which must be paid for the processing of the start and the end of a scope-defining construct, and for the processing of a declaration. It is not sure at all that in large modular systems this will be the case. The problem is not specific to the implementation of the Ada language; in the implementation of any language supporting safe separate compilation this problem will be met with when dealing with large and very large systems. Kamel et al [Kamel-84] discuss the use of other languages with large projects and the efficiency problems that occur. One of the cases reported caused 90

files to be opened at compile time and 11,700 symbols to be loaded.

Conradi [Conradi-85] discusses implementation paradigms for the handling of separate compilation. In particular the problem of the so-called *big inhale*, i.e. the large transitive import of the declarations in the environment, is dealt with. Although some remedies are given, they seem hardly suitable for handling the problem for languages with scope and visibility rules in the style of the Ada language. This problem is one of the set of problems that should be addressed in the near future.

4.6 Generic declarations and instantiations

One of the more interesting features of the Ada programming language is the capability to define *generic* language elements. Generic language elements are language elements that can be parameterized by types and subprograms as well as by objects. However, prior to using such elements they have to be instantiated, provided with actual parameters. The elements are in fact *code templates*, i.e. parameterized program modules. The assumption is that the use of these generic units may save programming time and increase the reuse of existing program components in the design and implementation of Ada programs.

Generic units are one of the four types of program units defined in the Ada language. Like a package, a generic unit has a specification and it usually has a body. However, a generic unit cannot be used directly, it is a template with which a normal subprogram or package can be produced. The power of the generic unit lies in its ability to be parameterized by objects, types, and subprograms.

At first sight, generic units are similar to macros. There are, however, some significant differences. Differences are: (i) a generic unit is semantically checked, even if it is never used, and (ii) a non-local reference in an instance of the generic unit is bound in the context of the declaration of the generic unit rather than in the context of its instantiation. An example of the use of generics is the package *text_io*. In this package (which is itself defined as a generic package) a number of generic packages is defined for input/output of specific types.

Relatively few papers have appeared on the implementation of generics. In fact only two could be traced. The first one is a description by Bray of the implementation of generics in the AIE compiler [Bray-83], the second is a description by Newton [Newton-86] of the implementation of Ada generics in the Ada+ compiler.

The two descriptions are examples of two different implementation strategies, the *sharing* approach and the *macro* approach. In the *sharing* approach, the code generated for a generic body is shared between all instances of the generic, regardless of the objects, types, and subprograms used in the instantiations. Bray describes the sharing of code for generic bodies for types that are similar, i.e. types that have the same characteristics.

The second approach, the *macro* approach is the one taken in the Ada+ compiler and in our Ada— compiler. The basic philosophy behind this approach is that for each instantiation of the generic unit a complete copy is made of the generic specification and its body - similar to macro expansion. These instantiations are used as ordinary language constructions.

The main advantage of the macro approach is its simplicity in the implementation. On the other hand, its main disadvantage is the increase of code size.

Static-semantic processing deals with both the declaration and the instantiation of generic templates.

A restriction in the Ada— generics is that a generic body that occurs as compilation unit is immediately preceded by its specification.

Generic declarations

The approach the Ada— compiler takes to performing static-semantic analysis on generics is to treat them as normal subprograms and packages, wrapped in a container that contains the declarations of their formals and that control visibility. Consider the example of a generic package

```
generic
  type T1 is private;
package p is
   a : T1;
end;
```

The tree structure representing the generic package is, schematically, given in figure 25.

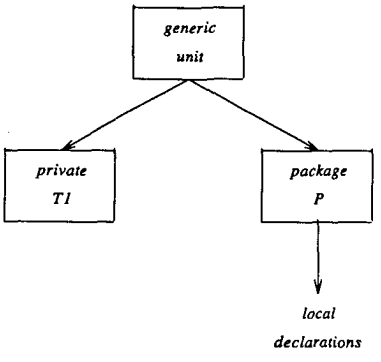


Figure 25. *Tree structure for generic unit*

The top node in the tree representation is a node indicating the unit being generic.

Producing instances

The second action that the compiler must take during semantic analysis is to expand the calls to generic units. In the Ada— compiler a choice was made for the macro model. The application of this model implies that for each generic instance the whole tree structure, representing the generic unit and its body, is instantiated by copying.

The applied algorithm is relatively straightforward:

1. The first step in processing an instantiation is to check that the name used in the instantiation denotes a generic unit of the appropriate type and that the actual parameters match the formal parameters. Ada allows formal parameters to denote values, variables, scalar types, array types, access types, unknown types and subprograms, all of which have their own matching rules. (Notice by the way that the matching rules are based on structural equivalence rather than name equivalence).
2. In this matching process the compiler creates a structure representing the actual-formal parameter correspondence and creates a copy of the specification and the body, with a suitable substitution of the formal parameters by the actual parameters. Creating the structure is by simply copying the tree of the generic specification.

Consider the example of the generic call:

```
package new_p is new p (integer);
```

The tree structure of the instantiated package is the structure corresponding to:

```
package new_p is
  a : integer;
end;
```

Checking instantiations

Due to the limitations of the generic specifications in the Ada— compiler circular instantiations cannot occur.

In the instantiated generic units checks must be performed on *contract violations.* Contract violations are caused by some combinations of generic formal private types, actual types, specifications, and bodies. They can happen because Ada allows the name of a generic formal private type to be used where the name of a constrained type is required, and allows an unconstrained type to be the actual for a generic formal private type, but does not allow combining the two.

Due to the restrictions on the generics imposed by the Ada— compiler, it is impossible to instantiate a unit before its body has been encountered. Therefore, contract violations can be checked during instantiation.

5. The Ada— compiler back end

5.1 Introduction

In this chapter the design and the implementation of the back end of the Ada— compiler is discussed. The contents of the chapter fall into four parts. First, the literature on the subject is discussed; then the design and implementation of the doublet model (the run-time data description model applied in the compiler). We discuss some other aspects of the lowering of the semantic level of the program representation, and finally we discuss the use of an existing code generator in the Ada— compiler.

In the process which must be performed by the compiler back end two important subprocesses can be identified: *lowering the semantic level* (code expansion) of the high-level intermediate program representation and *code generation.*

The input to the compiler back end, a high-level intermediate program representation, provides an almost completely machine-independent representation of source programs. A drawback of such a high-level intermediate representation is that the direct generation of target code constitutes a tremendous effort. By dividing this task into two subtasks, both subtasks are manageable. The output of the first phase (the expander) , generates another intermediate level which can be translated by a straightforward code generator. A second gain is that the number of target dependencies in the expander is so small that it can be parameterized on these dependencies. In their discussion on Lolita, the low-level intermediate representation for the Ada root compiler, Roubine et al [Roubine-82] point out that the size of bytes and words, alignment constraints, and peculiarities are specific target dependencies.

The two compiler back-end phases in the Ada— compiler, viz. the expander and the code generator, are implemented as separate programs. The expander implements a mapping of the high-level intermediate representation to the low-level intermediate representation. The code generator implements the mapping from this low-level intermediate representation to target assembler code.

The main function of the expander is to map constructs of the intermediate high-level program representation to semantically equivalent constructs in a low-level representation. In the expander, a number of assumptions is made about an *Ada virtual architecture*, a hypothetical architecture defining some characteristics of the final target machine. The expander maps intermediate tree structures onto this virtual machine.

The main functions of the expander are:

- declaration processing and storage allocation. Determining the layout of procedure activation records, determining the layout of complex data structures and assigning addresses and access functions to all entities having a counterpart at run time;
- expression translation. Mapping source-level operators such as e.g. "+", "*", but also operations as e.g. "." (selection), on appropriate operations or sequences of operations in the low-level intermediate code.

- Control structure translation; Mapping the source level control structures, e.g. a *for statement*, on appropriate low-level constructs in the low-level intermediate code;

The output of the expander is the program in two representations:

- a low-level program representation which is fed into the code generator for further processing in this compilation;
- a high-level representation (essentially the same as the input) which is annotated with addressing and storage information of the constituent objects and type declarations. This representation is fed into the program library for use in the compilation of other compilation units.

The structure of the expander is given in figure 26.

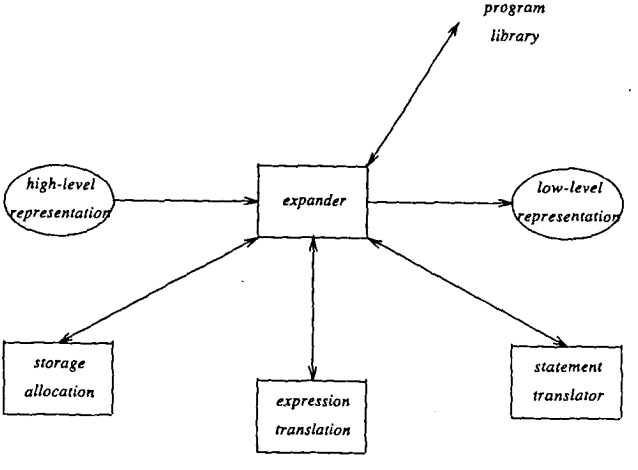


Figure 26. *Compiler expander structure*

The code generator is the second phase of the compiler back end. Its main function is to do code selection and register allocation for the real target machine. The design and the implementation of the code generator is to a large extent determined by the level and the structure of the intermediate program representation. The code generator of the portable C compiler variant on our local UNIX system was chosen as code generator for the Ada— compiler. Two optimizers exist that can be used in conjunction with this code generator, an intermediate code optimizer and an object code improver.

By chosing an existing code generator, an implicit choice is made for a low-level intermediate program representation. In this case, where a choice was made for the code

generator of the portable C compiler, the low-level intermediate program representation is tree structured. It is discussed in more detail in section 5.6.

In the remainder of this chapter various elements of the expander are discussed. The modelling of the code for expressions depends on the run-time data description model; section 5.3 contains a discussion on this run-time model and the generating of efficient code. The modelling of the implementation of exceptions and exception handling is discussed in section 5.4. Section 5.5 contains a discussion on the machine model and the lowering of the semantics of the high-level intermediate program representation (the *statement translator*). Section 5.6 is devoted to a discussion on the use of an existing code generator. No explicit attention is given to the storage allocator; its functionality is completely determined by the run-time data description model which is discussed in section 5.3.

5.2 Literature

A fair number of papers and descriptions on back ends (or code generators) for Ada language compilers has been published. The descriptions vary in their level of detail; some descriptions are just a few pages while others take several tens of pages. In this section we give a brief survey of the literature available.

One of the oldest descriptions of a back end for an Ada language compiler is the description of the back end of the Charette Ada language compiler. The back end is discussed in three papers. The first paper, [Rosenberg-80], discusses the translation of language constructs; it focuses, however, on MIL. MIL is a low-level language designed as a tool for language translation, *not* as a general purpose programming language. MIL programs only exist in Linear Graph Notation, a predecessor of IDL [Nestor-86]. Apparently code generators for the transformation of MIL programs existed.

The second paper [Hisgen-80] discusses in some detail the run-time representation of Ada variables and types. Some clever ideas from that description have been used in our descriptor model. In the run-time storage model rather complex descriptor structures have been used to describe the various data structures. These descriptors are based on the concept of dope vectors, i.e. each data object is described by a descriptor private to the data object.

The third paper, [Sherman-80b] discusses the final phase of the Charette Ada language compiler. It discusses, in particular, problems which were met within the implementation of subprograms, parameter passing, function-return values and exception handling. The discussion concentrates on VAX peculiarities in the support of operations; up-level referencing of variables and parameters, returning Ada objects from functions, and parameter sizes. Similar to our approach (to be discussed in section 5.5), their solution to the problem of composite function-return values is by retaining the activation record of the function, whose result is still in use. According to the authors, there exist cases where the simple stack model fails or where the underlying VAX hardware does not support the chosen model.

Some design issues of the European root-compiler back end are discussed by Teller et al [Teller-81]. The paper focuses on particular portability requirements for the compiler. In the paper technological consequences of these requirements are developed. The specific issues to be discussed are data allocation and the low-level intermediate language. The compiler model supported in the European root compiler has three major parts as was discussed in chapter 3. The second part, the expander, reads the intermediate high-level tree representation and generates a low-level intermediate tree representation. A main task of the expander is storage allocation, i.e. the connection of objects to a frame and the determination of a proper storage class for the objects. Some notes on the low-level intermediate representation are given in section 5.6.

The specification for the AIE Ada language compiler [AIE-82] contains a detailed discussion on the proposed back end. This back end consists of a number of phases, the names of the phases do remind of the names of the Bliss/11 compiler phases. Distinguished are:

— a flow analysis phase. In this phase extensive data and control flow analysis is performed to inspect whether optimizations are possible or not. The phase includes inter-procedural as well as intra-procedural analysis.

— VCODE, a virtual code generator to determine register usage. VCODE operates on a unit per unit base, where a unit is an Ada subprogram, a package, a task or an entry/accept body.

— TNBIND (Temporary name binding), a phase in which it is determined where each expression and object is to be computed; non-productive data moves can then be eliminated.

— CODEGEN, generating a doubly-linked list of target instructions. The job of CODEGEN is relatively straightforward due to the decisions taken in previous phases. The phase uses table-driven algorithms; it performs a reverse-execution-order walk so as to allow matching of the longest templates.

— FINAL, finalizing the job of code generation by resolving branch instructions and performing a standard set of peep-hole optimizations.

A detailed description of the code generator for the Karlsruhe Ada compiler is given by Jansohn and Landwehr [Jansohn-82]. The code generator takes two phases, a *middle end* and a *back end*. The middle end (in our model the *expander*) performs the subtasks of the transformations which are highly machine-independent. The back end (in our model the *code generator*) performs the machine-dependent subtasks, e.g. code selection, register allocation and assembly. The input of the middle end is a DIANA representation of the compilation units; the intermediate code between middle end and back end is the low-level language AIM.

The York compiler employs a three-stage code generator, briefly described by Briggs et al [Briggs-83a]. Although the code generator can be thought of as existing of a pipe-line of three programs, it is implemented as a single program. The stages in this program are a *planner*, an *expression-expander* and a *target-coder*. The basic tasks for the planner are to

register storage requirements and to process types and declarations. The expression-expander generates code directives for expressions. It is based on the classical Aho & Johnson dynamic-programming algorithms and the implementation is template-driven. Finally, the target coder produces machine code that will have the agreed semantic effect.

The code generator in the BreadBoard compiler is effectively discussed in two papers. Nowitz [Nowitz-82] discusses how C source code is generated as low-level intermediate program representation for a variety of constructs. The amount of C-code that is generated is rather large. The discussion by Nowitz shows that the translation of DIANA code to C code is straightforward. For each variable a pair is used similar to our doublet model. Rather than eliminating the doublets by optimization, they remain. As an example, for a simple assignment statement

```
x := 5;
```

the following C code is generated

```
reg [1]. v_value. int_v = 5;
reg [1]. v_type = (h1 + 2) -> v_type;
if (reg [1]. v_value. int_v < __VAR (0, 0) -> v_type. int_tp -> int_first
 ||reg [1]. v_value. int_v > __VAR (0, 0) -> v_type. int_tp -> int_last)
    { RAISE (CONSTRAINT_ERROR);}
__VAR (0, 0) -> v_value = reg [1]. v_value
```

The right hand side value is stored in a temporary register (*reg*); next a constraint check is performed, and if the result is satisfactory, the assignment is done.

Rubine [Rubine-82] discusses an hybrid Ada interpreter. This hybrid interpreter allows a module that is being interpreted to call a separately compiled module; it also allows a separately compiled module to call a module that is to be interpreted. Previously debugged pieces of a large program may be compiled and execute quickly when called.

A general description of the DDC compiler back end is given by Meiling et al [Meiling-83]. According to Meiling et al the back end of the compiler consists of four principal components. The first one is the generic instantiation handler. The second component is the transformer from IML_6 to IML_7; the latter is an intermediate tree-based language (for an example of IML_7 the reader is referred to section 5.7 of this thesis). In the compiler phase implemented by this component a few operations are broken down into elementary operations. The main task of this phase is to compute information guiding the code generation process.

The third component performs the major part of the code generation. The output of this component is a sequence of abstract A-code (AA-code) instructions. The fourth component produces the A-code. The transformation from AA-code to A-code involves expansion of AA-code instructions into one or more A-code instructions and creation of linkage information.

A brief description of the Ada+ code generator (targeted towards the Perq hardware) is given by Maddox et al [Maddox-86]. The description of the code generator consists of

two parts. The first part describes a run-time storage management scheme, the second part describes the mapping of various Ada language constructs. The run-time storage structure is heavily influenced by the Charette Ada compiler's run-time structure [Hisgen-80]. The scheme is essentially dope-vector based.

Recently, Kruchten [Kruchten-86] presented a doctoral thesis (in French) in which he discusses a virtual machine for the execution of compiled Ada programs. The machine was meant for the C-coded version of the Ada/Ed compiler. The contents of his thesis and the contents of this chapter have a certain amount of overlap.

Kirchgasner et al [Kirchgasner-82] discuss optimization techniques in the context of a so-called optimising Ada language compiler. Their report contains a number of useful tricks and transformations that are applicable under certain circumstances for Ada language compiler back ends. Apart from simple optimizations such as constant folding, optimizations are presented that require a large amount of data-flow analysis. Particularly interesting is of course the elimination of constraint checks. It turns out that a significant amount of time is spent with useless constraint checks, i.e. constraint checks for which it can be verified at compile time what their result will be.

Another paper on optimization techniques for Ada is presented by Winterstein et al [Winterstein-85]. In this initial study on a portable optimising compiler for a production-quality European KAPSE, a number of optimizations is presented. Particularly interesting areas seem to be the optimizations of run-time constraint checks, optimizations in the context of exceptions, optimizations in the presence of tasks, optimizations of generic program units and optimizations in the context of separate compilation.

A detailed study on the various optimization techniques (as discussed in the last two reports) is not in the scope of this chapter.

5.3 Describing data: the run-time data description model

Several proposals have been made, both in hardware and in software, to provide an adequate description for data structures in Ada programs at run time.

Bishop [Bishop-80] discusses a hardware implementation of dope-vector-like descriptors. Her ideas were based on the experience with descriptors for machines like the ICL 2900 series and the B6700 series.

Grovers et al [Grovers-80] discuss a virtual machine for Ada. Their virtual machine provides a repertoire of abstractions suitable for describing the semantics of the language. It seems, nevertheless, a fairly conventional P-code machine.

Hisgen et al [Hisgen-80] provide a rather detailed description of the run-time layout of variables and their descriptors as implemented in the Charette Ada language compiler. The scheme is strongly dope-vector-oriented; some factorization has been done on equal parts in descriptors. The prime idea of dope vectors, viz. each object having its private descriptor, remains intact. The same approach has been taken in the Ada+ compiler [Maddox-86].

Garlington [Garlington-81] defined and implemented - as part of a master's thesis assignment - a pseudo machine for the implementation of the Ada language. In his thesis he discusses a test translator (for a small subset of the Ada language) and the preliminary design of a pseudo machine architecture. The proposed architecture contains a number of operators particularly designed for the support of tasking. No special provisions were made to support the run-time description of data.

Other virtual machines have been defined as well, most of them providing some support for the run-time description of data and generating efficient code for accessing elements in composite objects. At least one architecture was defined and implemented in hardware. The NOKIA MPS-10, a capability-based architecture (described by Lahtinen [Lahtinen-82]), is an example.

Kruchten describes a complete virtual machine for Ada. The model that is used for the run-time representation of type information has a resemblance with the model used in the Ada— compiler. The structuring of the descriptors is similar. The fact that Kruchten uses a tagged architecture seems to suggest a different approach than the approach taken in the Ada— compiler.

We felt that an implementation based on dope vectors would become complex. The dope-vector approach may be very useful for languages like Algol 60, i.e. languages where the structural properties of variables are specified explicitly in their declarations. It has some disadvantages for the implementation of the Ada language:

1. variable declarations may share a type or subtype; it is virtually impossible, however, to share descriptors. Even for the description of array elements tricks have to be invented to prevent the construction of a descriptor private to each array element [Goor-81];
2. complex data structures are often implemented as tree structures. Tree structures are hard to implement, hard to manage and hard to operate on.

Unlike other implementers who have chosen for data description models based on dope vectors, [Hisgen-80], [Dismukes-84], we decided to avoid the potential problems that could come up with the use of such a model. Therefore, an alternative model, the doublet model, was defined and implemented. In the remainder of this section the model is described and an outline of the applied implementation strategy is given.

5.3.1 Descriptors and the doublet model

The doublet model defines that for each elaboration of a type definition, a subtype definition or a constraint, a *descriptor* implementing the type definition, the subtype definition, or the constraint is constructed. A *reference* from one (sub)type definition or constraint to another (sub)type definition is implemented by a pointer to the descriptor implementing the latter, although there are some exceptions on practical grounds. The descriptor for *b* in

```
subtype b is c (10);
```

thus contains a pointer to a descriptor for *c*. The renaming of subtype declarations form

an exception to these rules. Consider the subtype definition to rename the existing type or subtype definition for *integer*.

subtype *int* **is** *integer;*

No separate descriptor is constructed for *int*; *int* and *integer* share the same descriptor.

Data in a descriptor should allow any piece of required information to be computed. Thus, the data stored in a descriptor should depend on the set of operations to be performed on that descriptor. Descriptors are used for constraint checking, they are also used to support the construction of other descriptors, for the generation of values and for the computation of addresses and descriptors of subcomponents in composite objects and values. For example, for the construction of *b*'s descriptor, with *b* as above, data in *c*'s descriptor is used.

A variable is characterized by a *doublet*. A doublet consists of a pair of pointers: one pointer to a location in which the value of the variable is stored, the other one to a description of the logical structure imposed on the bits of the location and the constraints put on values for that location. For any object, whether it be declared explicitly or required for storage of intermediate results, a doublet exists by definition. A schematic representation of a doublet is given in figure 27.

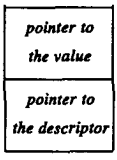


Figure 27. *A doublet*

By convention, the value of a variable is encoded as a sequence of bits, irrespective of the logical structuring of the value and without any contained description. Doublets do not contain descriptors, they only refer to them; descriptors can be shared by all doublets implementing objects of the same type, subtype or constraint.

Different descriptors exist for different classes of types and subtypes. There are descriptors for scalar types and subtypes, for array types, for array subtypes, for record types, for record subtypes, for access types, for access subtypes and for task types.

Descriptors are implemented as (linear) sequences of machine words. They are not viewed as atomic entities; it must be possible to access their individual components. By convention, the first word in any descriptor is a *tag field* containing a *tag*, an identification of the kind of the descriptor. Identification of the kind of operator is required at run time for operations like the default initialization of record components and the testing of equality of composite values; operations that are completely performed by run-time

routines.

Definitions of scalar and numeric types depend on other types, e.g. in

subtype *shorter* **is** *int* **range** *1 .. 11;*

shorter depends on *int*. Nevertheless, a descriptor for a numeric or scalar type does not contain a pointer to other descriptors, simply because there is no need for such a pointer. Such a descriptor contains a *tag*, an indication of the size of the values of the type or subtype and an encoding of the range of values applicable for the type or subtype. The descriptor for subtype *shorter*, $D_{shorter}$ for short, is given schematically in figure 28.

The descriptor for an array type contains a *tag*, pointers to descriptors for the index types, a pointer to the element type and the element size. The element size is the number of bytes occupied by a single array element. A schematical description of a descriptor for a one dimensional array type, *aint*, where *aint* is defined as

type *aint* **is array** *(int* **range** <> *)* **of** *int;*

is also given in figure 28.

An array variable has always some constrained (sub)type and is therefore described by an array subtype descriptor. For each elaboration of an array-subtype definition or a constraint on an array type, an array-subtype descriptor is constructed. An array-subtype descriptor contains a *tag*, a pointer to the descriptor of the type being constrained, and the constraint values. The descriptor for *saint*, where *saint* is defined as

subtype *saint* **is** *aint (index_constraint);*

is also given in figure 28.

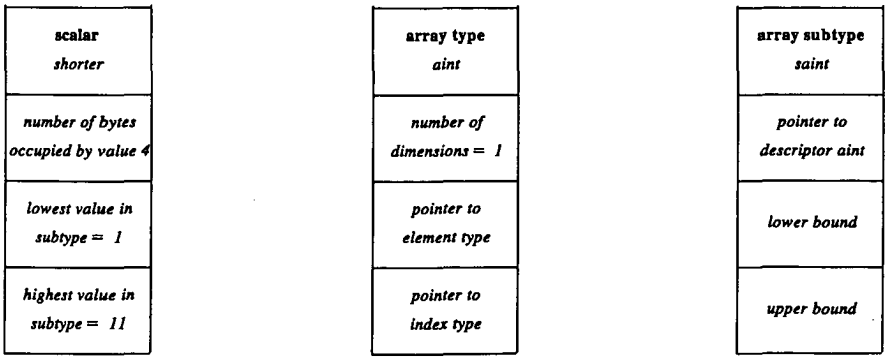


Figure 28. *Descriptors for shorter, aint and saint*

A record type contains discriminants and ordinary record components. A record-type

descriptor contains a *tag* and pointers to descriptors for the types of the discriminants and the types of the other record components. Whenever necessary, the descriptor for the constraint of a record component is constructed during the elaboration of the record type. For example, in

```
type c_rec is record
   rc1: aint (1 .. expression);
end record;
```

an array subtype descriptor, implementing the constraint

```
aint (1 .. expression)
```

is constructed during the elaboration of *c_rec*'s definition. Constraints for record components may depend, however, on discriminant values of the enclosing record type. (A record component with such a constraint is called a *discriminant-dependent record component.*) Consider the example

```
type u_rec (d : int) is
record
   rc1 : aint (M .. d);
   rc2 : saint;
end record;
```

The component *rc1* is a discriminant-dependent record component; the constraint

```
aint (M .. d)
```

depends on *M*, a constant or an expression, and on *d*, a discriminant of the enclosing record type. *rc1*'s description cannot be constructed until the discriminant values are known, i.e. a value is provided for *d*. In general, a descriptor for a discriminant-dependent record component cannot be made during elaboration of the enclosing record type. During the elaboration of the record type, for each discriminant-dependent constraint a *call block*† is made instead. The descriptor of the enclosing record type is constructed as usual. For each discriminant-dependent component, however, it contains a pointer to such a call block. A call block provides the data for the construction of the actual descriptor once the discriminant values are known. It takes the form of an ordinary descriptor. For each discriminant it contains an encoding of the place where the discriminant value can be found in the enclosing record type. Furthermore, it contains a *tag*, a pointer to the descriptor of the type to be constrained and the discriminant values that can be computed during the elaboration of the enclosing record type.

As an example, on the elaboration of *u_rec* the descriptor implementing the type definition *u_rec* is made. It contains pointers to descriptors describing the (sub)types of the discriminant *d* and the component *rc2* and a pointer to a call block for component *rc1*. The call block for the constraint on *aint* in the definition of the record type *u_rec* contains a *tag*, a pointer to the descriptor of *aint*, the value of *M* and an indication where the value

† Call blocks originated in the Charette Ada language compiler

of the discriminant *d* can be found. A schematical overview of the whole tree of descriptors for the record type *u_rec* and its components is given in figure 29.

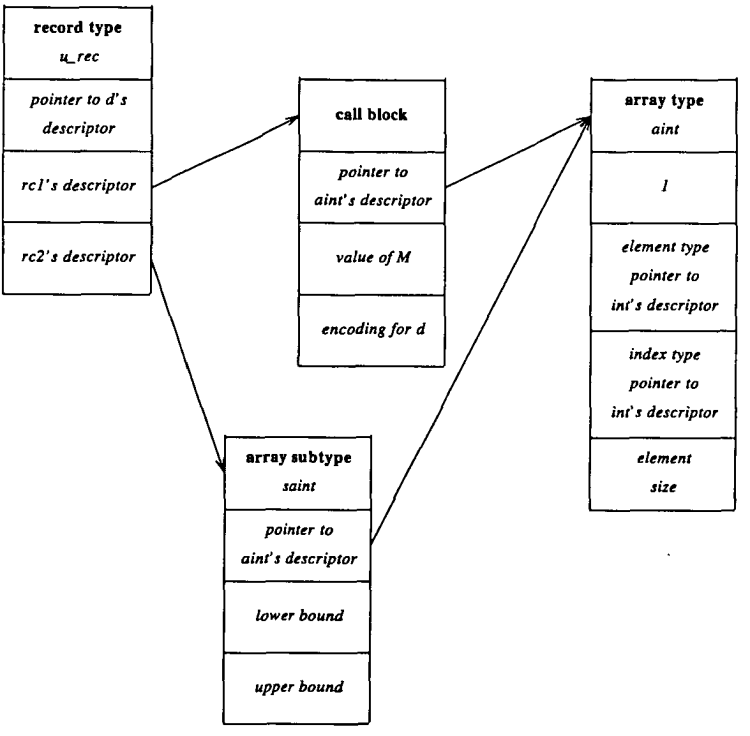


Figure 29. *Tree of descriptors for record type u_rec*

For a discriminantless record type, the descriptor itself contains (together with the descriptors for the components) sufficient data to describe any variable or value of the type. For records with discriminants a distinction has to be made between a record-type descriptor, implementing the record-type definition, and a record-subtype descriptor, implementing a constraint on that record type. For each elaboration of a record-subtype definition or a constraint on a record type a record-subtype descriptor is constructed. A record-subtype descriptor contains a *tag*, a pointer to the record type being constrained, and the discriminant values. A complication is that on the elaboration of a constraint on a record type with discriminant-dependent components not only a record-subtype descriptor must be made; for each of the discriminant components a descriptor must be constructed as well. The descriptor for a discriminant-dependent component can be made from the

discriminant values and the data in the call block for that component. For each elaboration of a constraint on the record type such a descriptor is constructed; it is kept private to the subtype descriptor for the record. Consider

subtype *su_rec* **is** *u_rec (expression);*

The elaboration of the definition of *su_rec* results in a descriptor for *su_rec* with attached to it a private descriptor for the *rc1* component; schematically given in figure 30.

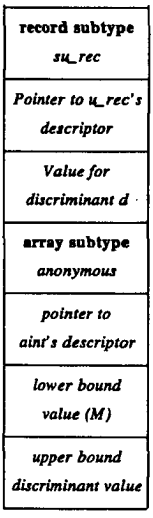


Figure 30. *Subtype descriptor for su_rec with private component descriptor*

The offset of *rc1*'s private descriptor is computed at compile time. For each discriminant-dependent component such an offset is known at compile time; the value of this offset is stored in an extra component (*cb_offset*) of the call-block structure.

An *access* type describes access values rather than the access objects. In particular, it does not describe structural properties of the access objects. The purpose of constraining an access type is to create a subset of the values belonging to the access type, not subsetting the accessed type. Consider the following type and subtype definitions

```
type u_rec2 (a: int);

type au_rec2 is access u_rec2 (expression);

type u_rec2 (a: int) is record
  p1 : int;
  p2 : au_rec2;
end record;
```

On the elaboration of the definition of *au_rec2* only an incomplete type definition of *u_rec2* has been elaborated; a descriptor for *au_rec2* does not yet exist. Therefore, it is not possible at this point in the elaboration, to construct a descriptor for the constraint

```
u_rec2 (expression)
```

The constraint in the definition of *au_rec2* implies that any object referred to by an access value of type *au_rec2* obeys the constraint. Another semantics-preserving interpretation of the definition is to see *au_rec2* as a subtype, constraining an implicitly declared anonymous access type, i.e.

```
type anonymous is access u_rec2;

subtype au_rec2 is anonymous (expression);
```

Since the descriptor implementing an access type does not have to describe structural properties of the access objects, there is no need for a pointer in an access type descriptor to a descriptor for the access objects. Therefore, even if a descriptor for *u_rec2* does not exist yet, an access type descriptor implementing *anonymous* can be constructed on the elaboration of the construct. Furthermore, a subtype descriptor for *au_rec2*, constraining *anonymous*, can be constructed. Any access type carrying a constraint on an incomplete type is interpreted this way.

A descriptor for an access type contains a *tag*. Furthermore, if the accessed type can be constrained, it contains pointers to descriptors for either the discriminant types or the index types. A descriptor for an access subtype contains a *tag*, a pointer to the access type being constrained and the discriminant values. A schematical representation of descriptors for the type *anonymous* and the subtype *au_rec2* are given in figure 31.

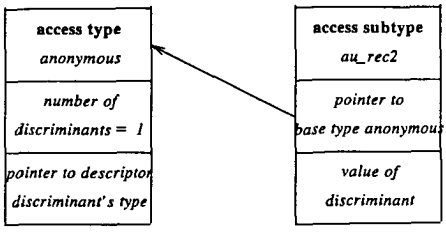


Figure 31. *Descriptors for anonymous and au_rec2*

Tasks are program components which may be executed in parallel. Tasks appear as values of task types which cannot be constrained. Task-type descriptors do not contain pointers to other descriptors. A task-type descriptor contains data to facilitate the construction of task values. It contains a *tag*, a component indicating the static priority of tasks of this type, a component containing the code address of the task and a component containing the number of entries of the task type. A schematical description of a task type descriptor is given in figure 32.

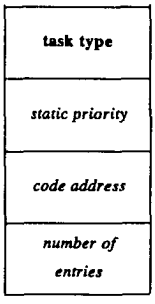


Figure 32. *A task type descriptor*

A task value contains all data required for the execution of the task. Given a task *type* descriptor, such a task value can be constructed.

5.3.2 Component addressing

Given the address of an array value, the index values and an array subtype descriptor, computing the address of an array element is almost trivial.

To compute the address of a record component, however, several cases, varying in complexity, have to be distinguished. Only Pascal-like records have components with statically known sizes and, therefore, known offsets. To compute the offset of a component in other kinds of records an *offset table*, implementing a mapping from components to offsets, is attached to either the type or the subtype descriptor. Two cases can be distinguished:

1. records with component constraints that are not dependent on discriminants of the enclosing record type. Component offsets can be computed during the elaboration of the record-type definition. An offset table is attached to the record's type descriptor and initialized during elaboration of the record type.
2. records with component constraints that are dependent on discriminants of the enclosing record type. Component offsets may differ from subtype to subtype. An offset table is attached to the subtype descriptor and initialized during the run-time elaboration of the record subtype.

The definition of *u_rec*, given earlier, is a case in point. The constraint on *aint* for the component *rc1*,

```
rc1 : aint (M .. d);
```

depends on *d*, a discriminant of the enclosing record type. The size of the *rc1* component can be determined at the run-time elaboration of the record subtype; an offset table is attached to the subtype descriptor. If *x* is a variable declared as

```
x : su_rec;
```

then the address of *x*. *rc2* can be computed by taking the address of *x* and adding the offset for *rc2*, found in the offset table attached to *su_rec's* descriptor.

Record-type definitions may contain variant parts, each of which may contain variant parts as well; a record type with variants can be seen as a tree structure. Descriptors for record types with variants and for subtypes of these record types carry data to support the verification of a record-component selection.

The paths through any record that contains variants can be numbered sequentially. Records without variants have, by definition, a single path numbered 1. For each subtype of the record type, the path (and therefore the path number) is fixed once the discriminant values are known. A path number only needs to be computed when the constraints are applied, i.e. on the elaboration of a subtype or a constraint. The path number can therefore be kept in the subtype descriptor. Consider the record type *rr* below.

```
type rr (d1, d2, int) is
record
  f1, f2 : int;
  case d1 is
    when -100 .. -1 =>
           f3, f4 : int;
    when 0      =>
           f5    : integer;
         case d2 is
           when -1000 .. -1 =>
                    f6    : integer;
           when others     =>
                    f7, f8, f9 : integer;
         end case;
    when others    => null;
  end case;
end record;
```

The paths through the record type can be numbered and a relation between the components and the various path numbers can be determined:

- path number 1: *d1*, *d2*, *f1*, *f2* and *f4*;
- path number 2: *d1*, *d2*, *f1*, *f2*, *f5*, and *f6*;
- path number 3: *d1*, *d2*, *f1*, *f2*, *f5*, *f7 f8* and *f9*;
- path number 4: *d1*, *d2*, *f1*, and *f2*.

For each component a *range* of path numbers can be specified within which the component can be selected:

- *d1*, *d2*, *f1* and *f2* may be selected in path numbers 1 .. 4;
- *f3* and *f4* may be selected in path number 1 only;
- *f5* may be selected in path numbers 2 .. 3;
- *f6* may be selected in path number 2;
- *f7*, *f8* and *f9* may be selected in path number 3.

Verification of the selection of a record component can be expressed as a range check†. The path number of the record subtype is matched against the range of path numbers of the component being selected. The computation of an offset, with a check on the legality of the selection, can be expressed as a piece of Ada program. For example, the computation for the offset of *f5* reads:

† It was found that in the proposed AIE compiler [AIE-82] the same principle has been proposed for checking variants.

```
if path not in 2 .. 3
then
  raise constraint_error;
else
  offset := 2 * integer' size;
end if;
```

path stands for code to fetch the path number in the descriptor for the record value in which selection is going to take place. To allow the computation of the path number for subtypes of a given record type, the record-type descriptor contains the entry point of a *path function.* This path function yields a path number when applied to discriminant values.

An interesting topic is the question which data should be available in descriptors to support complex run-time operations, such as tests on equality and default initialization of record components. These operations are performed by run-time routines operating on descriptors. Since the routines iterate at run time over the record components, the data in the descriptor should be sufficient to allow such an iteration to be made. For ordinary records, the data in the descriptor is sufficient. For each of the record components the descriptor contains a pointer to the record component's descriptor. The data in the descriptor need some extension for variant records. Not all components for which the record-type descriptor contains a pointer to the descriptor are in the variant on which the iteration has to take place. Therefore, in case of a variant record the record-type descriptor is extended with a *range table.* This table contains for each record component the range of allowable path numbers. Given the path number of a record subtype, it can be determined whether a given record component is in the variant of the record subtype or not.

Run-time computation of the offset of a record component is sometimes eased by the availability of an offset table. Without an offset table this computation is expensive. Adding an offset table to every record type or subtype descriptor may turn out to be the most efficient strategy. Run-time computation of a record component's offset, without having an offset table at hand is done by adding the sizes of all preceding components to each other. Component sizes and alignment requirements can be found, directly or indirectly, in the descriptors for the component (sub)types. These descriptors can be found either through the record-type descriptor or the record-subtype descriptor.

A descriptor for a record type, with components to be initialized, contains a special table. For each such component the table contains the entry point of the initialization code. It can be determined for any given record component whether it has an initializing expression or not, simply by iterating over the record's components. The descriptor for a component to be initialized can be found through the record type descriptor or, if the component is a discriminant-dependent one, in the record subtype descriptor. Finally, the evaluation of the initialization code requires the environment, in which the record type definition was elaborated, to be restored. Therefore, the record type descriptor contains a link to the appropriate environment; this link is used to re-establish the environment that was in force prior to the evaluation of the initialization code.

5.3.3 An implementation of the doublet model

The doublet model, though elegant and simple, is rather inefficient when implemented literally. The model has been implemented as part of an interpreter for DAS by Niels Bogstad and Albert Hartveld [Bogstad-83]. In this subsection the approach to the current implementation is presented.

The basic idea behind the implementation of the doublet model is to have a compile-time simulation of the run-time behaviour of doublets. In programs in a compiled language like the Ada language, references to a variable or an object are almost always translated under compiler control. Whenever possible, a compiler generates code to access the value or the descriptor involved directly. If, for a given construct yielding a doublet by strictly adhering to the model, no run-time references are left to the doublet itself, construction at run time of that doublet is not required. In a vast majority of cases the compiler is able to circumvent doublet construction; in only a few cases doublets have to be constructed at run time.

Simulation of doublets is done by two functions, *Descriptor* and *Value*. These functions operate on virtually any construct in the Ada programming language which yields a doublet according to the doublet model.

Descriptor — When applied to an Ada construct, *Descriptor* yields access code for the descriptor of that construct. When applied to a type, to a subtype or to a constraint, *Descriptor*, by convention, yields code to access the descriptor implementing the type, subtype or constraint.

Value — When applied to an Ada construct, *Value* yields either code to compute the value or code to access the value of that construct. The function *Value* is called for the translation of any reference to the value of a construct which yields a doublet. For any reference to the descriptor for a construct yielding a doublet, which is to be translated under compiler control, the function *Descriptor* is called with as parameter that construct.

Consider the example of the assignment statement

```
a := expression;
```

The compiler generates code for the left hand side of the construct, for the right hand side of the construct and for the assignment itself. It applies *Value* to *a*,

```
Value (a)
```

to generate code to access the target location. Then it applies the function *Value* to the right hand side expression to generate code to evaluate the expression and to access the result value. Finally, it generates code to perform the assignment.

The functions *Descriptor* and *Value* have an immediate counterpart in the implementation. In the implementation they operate, however, on an intermediate tree representation rather than on source level constructs. Their outputs are prefix-encoded expression trees, encoding of computations yielding the address of either the value or the descriptor of the construct passed as argument. The resulting computations are presented

as prefix encoded expression trees; similar to expression trees appearing as intermediate code in the portable C compiler [Johnson-78]. It was already reported that the code generator in our compiler is borrowed from an existing variant of the portable C compiler.

The operators used in the expression trees are derived from the set of operators in the C language. For almost any operator in C there is a corresponding operator in the intermediate representation. In addition to the usual arithmetical operators an *assign* operator, a *call* operator and a *dereferencing* operator are supported. In this paper symbolic names are used to denote the operators. As an example, the C expression

a = b + c

is encoded as

assign
  access code for a
  plus
    access code to the value of b
    access code to the value of c

(For reasons of readability, trees are written indented with a single operator per line.)

The *call* operator takes two parameters: the first parameter is an expression yielding the entry point of the function to be called, the second parameter is a list of function arguments. Arguments are separated by a binary *parameter comma* operator. The C expression

f(3, 4)

is encoded as

call
  f
  parameter comma
    3
    4

Finally, a monadic dereferencing operator is indicated by the operator *deref*. Consider the example of accessing a local variable on an M68000-based machine. The base pointer is stored in *a6*. The offset relative to the base pointer is indicated *Offset*. The encoding of the expression tree to access the contents of this location reads

deref
  plus
    a6
    Offset

The use of prefix code is not essential. It was reported earlier that an experiment was carried out in which EM code, a postfix stack-oriented code, was used.

We discuss the functions *Value* and *Descriptor* by describing the code they generate for some of the primary name constructs. It will be shown that for most constructs the use of these functions leaves no references to doublets at run time, eliminating the need to actually build these doublets. The constructs chosen are those which frequently appear in expressions: constants, variables and name constructs as indexing, selection, slicing, dereferencing, and calling. The code will be expressed in the form of templates, built up from low-level intermediate operators, constants and further calls to the functions *Descriptor* and *Value*. The functions are in some sense described recursively.

For each constant, e.g. *10* or *red*, both the value and the type are known at compile time and there is no need to build a doublet for any constant. The functional behaviour of the functions can be described as follows:

Descriptor yields the access code of the known descriptor of the constant,

Value yields the access code to a representation of the value of the constant.

As an example, *Descriptor (3)* e.g. yields the address of $D_{integer}$, *Value (3)* yields the constant value *3*.

For a declared object, e.g. A in

```
A : saint;
```

(with *saint* as given earlier, page 113) the descriptor is known to be the one implementing the (sub)type or constraint given in the object declaration.

```
Descriptor (A)
```

yields a structure, evaluation of which yields the address of D_{saint}. The actual code to address the descriptor depends on its storage class (i.e. the area in storage where the descriptor is stored). Storage classes that can be distinguished are: global, (addressable through a label) or addressable as some stack object (either local or through a static chain). For ordinary variables, for which all references to their location and to their descriptor are translated under compiler control, there is no need to construct a doublet at run time. The functions *Descriptor* and *Value* are always capable of generating code without having to look into a doublet. The code resulting from applying *Value* to some variable, e.g.

```
Value (a)
```

only depends on the storage class of the object. For variables (and temporaries) the distinguished storage classes are: global, addressable as stack object and addressable as heap object.

In the Ada language, names may denote declared entities, objects designated by access values, subcomponents and slices of values. According to the doublet model, for each intermediate result a doublet exists by definition; it is shown that, in practice, it is seldom required to have such a doublet available at run time.

Consider the name construct

name (expression)

where *name* is a construct delivering an array value and *expression* is an indexing expression. The address of the indexed element is computed by adding an offset to the base of the value of the array being indexed. The formula for the computation of the offset is well known:

*offset = (expression - lwb) * element_size*

In this formula *lwb* stands for the lower bound of the array while *element_size* stands for the element size, the number of bytes per array element. The structure of the code to be generated for addressing an indexed component, i.e. *Value (name (expression))*, can be expressed as a template containing operators, constants and recursive calls to *Descriptor* and *Value*. The template reads

```
plus
  Value (name)
  mul
    sub
      code for expression
      deref
        plus
          Descriptor (name)
          lower_bound
    deref
      plus
        Descriptor (basetype (name))
        element_size
```

The function *basetype* computes, at compile time, the array type (not the subtype) of *name*. The array-type descriptor has a component at offset *element_size* that contains the size of an array element. The constant *lower_bound* is an offset of the component within the array-subtype descriptor containing the lower bound. An optimization that is often applicable is to replace the code for fetching the element size from the descriptor by the element size itself. For scalar array elements in particular, the element size is known at compile time.

Two cases have to be distinguished for record component selection: selection of a discriminant and selection of an ordinary component. If, in

name. component

name is a construct delivering a record value and *component* is the name of a discriminant of the record type, the requested value can be found in the descriptor of *name* rather than in its value. Assume that the discriminant's offset within the descriptor is Id_discr_i. The code to be generated for addressing the value, i.e.

Value (name. component)

can be expressed as the template

plus
  Descriptor (*name*)
  Id_discr$_i$

Discriminants always have scalar (sub)types, the descriptors for these (sub)types have compile-time-known addresses. If the type of *name. component* is *st*, the code to be generated for

Descriptor (name. component)

is the address of

D_{st}

If *component* is an ordinary record component, the code to be generated for

Value (name. component)

can be expressed as the template

plus
  Value (name)
  some_offset

some_offset is either a constant offset or it stands for a sequence of operations by which the offset from an offset table is fetched. For an offset table attached to the subtype descriptor and an offset *offset*, relative to the start of that descriptor, the code template is

plus
  Value (name)
  deref
    plus
      Descriptor (name)
      offset

If the type or subtype of a record component is known, code to address the descriptor of the record component's type or subtype is the address of the descriptor. If the type of *name. component* is *some_type*, the code to be generated for

Descriptor (name. component)

is simply code to address

D_{some_type}

For discriminant-dependent record components, computing of the address of the descriptor is more complicated. It has already been explained that descriptors for such components are kept private to descriptors which provide the constraint values, i.e. descriptors for subtypes of the enclosing record type. If *name* is a construct yielding a record value and *component* is the name of a discriminant-dependent component of that record, then the code template for

Descriptor (name. component)

is

plus
  Descriptor (name)
  offset

In this template *offset* is the, compile-time-known, offset of the private descriptor. It can be recomputed dynamically by interpreting the *cb_offset* field in the call block from which the descriptor of the selected component originates.

In the cases presented so far, a descriptor always existed and was addressable, independently of whether the value had been computed or not. There are several cases where the existence of the descriptor is related to the existence of the value and where doublets *are* required at run time. This happens with functions specified to return a value of an unconstrained type and with access values specified to access objects of some unconstrained type.

Consider the function

function *f (a: integer)* **return** *u_rec;*

where *u_rec* is the unconstrained record type given earlier (page 114). Since the constraints of the return value cannot be inferred from the specified return type, the function has to return a complete doublet. Some care has to be taken to ensure that, prior to evaluating any code accessing the descriptor, the function has actually been evaluated. The computation for

Value (f (10))

is

deref
  plus
    call
      code for entry point of F
      10
    offset in doublet

The function call yields a doublet. The component at offset *offset in doublet* contains a pointer to the actual return value. The computation for addressing the descriptor of the returned entity is similar, the offset in the doublet, however, is different. Notice the difference between this case and the case where the function result is specified to be constrained, e.g. in

function *G* **return** *su_rec;*

with *su_rec* a constrained subtype of *u_rec* (page 116). The function returns simply (a pointer to) a value. The descriptor of the return value is addressable and accessible independently of the computation of the value. The computation resulting from the call

Descriptor (G)

yields the address of

D_{su_rec}

The second case in point is dereferencing. Consider the name construct

***name*. all**

where *name* is a construct delivering an access value e.g. *au_rec*, where *au_rec* is defined as

type *au_rec* **is access** *u_rec;*

The access object itself has to provide the constraints, they cannot be inferred from the access type. Similarly to return values of functions with an unconstrained return type, the access object is implemented as a doublet. The computation resulting from

Value (name. all)

can then be expressed as the template

deref
  plus
    Value (name)
    offset in doublet

The computation for *Value (name)* yields the address of a doublet. An element of the doublet, at offset *offset in doublet*, contains a pointer to the value. The template describing the computation for

Descriptor (name. all)

is similar; the difference is the offset used in the computation.

If an access type accesses a constrained type or subtype, then the descriptor of the access object is known a priori. In such a case, *Descriptor (name. all)* can be computed without having to refer to the access object. Consider

type *a_integer* **is access** *integer;*
name : a_integer;

...

Descriptor (name. all) is known to yield the address of $D_{integer}$, the descriptor implementing *integer*. In those cases where the descriptor of the accessed object is known, the accessed object can be implemented as a simple pointer to a value rather than as a doublet.

Slicing is a special case. After slicing has been applied, a part of the value to which slicing is applied is effectively renamed. The description for the slice, however, is a new one. Consider the slice:

name (i .. j)

where *name* is a construct yielding a one-dimensional array value. The address of the slice can be computed in terms of the address of the value of *name* and an offset, determined by the lower bound of the range. Since the subtype of a slice is not known a priori, a descriptor for the slice is to be constructed as part of the slice operation. Its address is put in a temporary location. Construction of a whole doublet is unnecessary. Addressing the descriptor, i.e. computing

Descriptor (name (i .. j))

is then simple. Computation of the address of the value of the slice, i.e. applying *Value* to the name construct

Value (name (i .. j))

yields the code template:

```
plus
  Value (name)
  mul
    sub
      access code for i
      lower_bound
    element_size
```

lower_bound is either a compile-time-known value or it stands for a computation to fetch the lower bound of the array from which the slice is taken.

element_size is either a compile-time-known value or it stands for a computation to fetch the element size from the slice's array-type descriptor.

5.3.4 An example

The process of code generation is exemplified by applying the functions *Value* and *Descriptor* to the assignment statements in the following Ada program.

```
procedure test is
  y : constant integer := 10;
  subtype int is integer;

  type aint is array (int range <> ) of int;
  type u_rec (a : int) is record
    x : aint (1 .. a);
  end record;

  subtype su_rec is u_rec (y);
  type au_rec is access  u_rec;
  type asu_rec is access su_rec;

  function f (a : int) return au_rec is
  begin
    -- a function body returning a value of type au_rec
  end f;

  function g (a : int) return asu_rec is
  begin
    -- a function body returning a value of type asu_rec
  end g;

begin
  f (y) . x (y) := 1;
  g (y) . x (y) := 1;
end;
```

The first assignment statement in the program has a rather complex name construct as its left hand element. In this construct the function *f* is called first. The return value, an access value accessing a record object of type *u_rec*, is then implicitly dereferenced and the *x* component is selected. The target location of the assignment is finally computed by indexing the *y*-th element in this *x* component.

Some remarks can be made about this program.

- *test* is assumed to be a procedure on the main level, i.e. a procedure without enclosing procedure.
- Code being generated for an M68000 based machine, *a6* is used as a base register; local variables and temporaries are addressed relative to this base register. In the intermediate representation the shorthand

  Offset(a6)

  is used instead of

```
deref
  plus
    a6
    Offset
```

- With each local function or procedure, a local variable is associated in the enclosing function or procedure (see also section 5.5). On elaboration of the specification of a function or procedure, the entry address of an error routine is stored in the associated local variable. When the body of the function or procedure is elaborated, the entry address of that function or procedure is stored in that local variable. Any call to the function or procedure will take the contents of the associated local variable as entry point. So, when a function or procedure is called prior to the elaboration of its body, an error routine will be called. In the program which serves as an example the local variable in *test* in which the entry address for *f* is stored has a compiler generated offset -32.
- Since *f* is local to the procedure *test*, a static link is required for addressing elements in its environment. The static link is passed as the first parameter in the call.
- The compiler generates various constants that appear in the generated code. Constants that are generated by the compiler in this example are: 4, 16, 28, 0, -32, -40.

It can be inferred at compile time that in the assignment statement

```
f (y). x (y) := 1;
```

the source value meets the constraints of the target location. No constraint check will be generated: the code to be generated therefore is code for a simple assignment.

Consider the first step in the code generation process the template for the assignment

```
assign
  Value (f (y). x (y))
  1
```

The code to access the target location of the assignment is generated by applying the function *Value* to the construct *f (y) . x (y).* Substitution of the constants and the parameters for the calls to *Descriptor* and *Value* in the template for the value of an indexing construct yields:

```
plus
  Value (f (y). x)
  mul
    sub
      10
      deref
        plus
          Descriptor (f (y). x)
          16
    4
```

The value *10* is the known constant value of *y*; the value *16* is the implementation determined offset of the component in the array-subtype descriptor, yielding the lower bound of the array. Finally, the value *4* is the number of bytes per array element known at compile time. The descriptor, for which access code is to be generated in this computation, is that of a discriminant-dependent record component. It can be found on the implementation-dependent offset *28*, relative to the base of the descriptor for the object of the enclosing record type. The call

```
Descriptor (f (y). x)
```

then expands to

```
deref
  plus
    Descriptor (f (y). all)
    28
```

The structure of the code for the computation of the address of the *x* component of the value accessed by *f*'s return value is straightforward. The offset of the *x* component has to be added to the address of the record value. Since *x* is the first component of the record (recall that discriminant values are kept in the descriptor of the value), this offset is *0*; the call

```
Value (f (y). x)
```

expands to

```
plus
  Value (f (y). all)
  0
```

The function *f* returns an access value; this value is implemented as a pointer to a doublet-like structure, given here as C structure

```
struct doublet{
      char *descr;          /* pointer to descriptor    */
      char val [VALSIZE];   /* storage for value        */
};
```

The first element is a pointer to a descriptor. The second element contains the value of the access object rather than a pointer to the access object. For each object created by an allocator *VALSIZE* has an appropriate value.

The offset of the *val* component in this doublet structure is *4*; the code to be generated for

```
Value (f (y). all)
```

then expands to

```
plus
  Value (f (y))
  4
```

Similarly, the offset of the *descr* component of the structure is *0*, the code to be generated for

```
Descriptor (f (y). all)
```

expands to

```
deref
  Value (f (y))
```

Computing the value of the function twice, once for the computation of an access path to a value, once more for the computation of the access path to a descriptor, as suggested above, should be avoided. The compiler detects situations where the value (descriptor) of a complex computation is used more than once in a single expression. It generates code to store a pointer to the value (descriptor) in a temporary location in such a way that this temporary result can be referred to in subsequent computations for the same value (descriptor) in the same expression. In this particular case a compiler-generated temporary location at *-40 (a6)* is used for that purpose. The code to be generated for the function call is

```
assign
  -40 (a6)
  call
    -32 (a6)
    parameter comma
      10
      a6
```

If we do not take this code for the function call into account, the resulting code for the assignment becomes

```
assign
  plus
    plus
      plus
        -40 (a6)
        4
      0
    mul
      sub
        10
        deref
          plus
            plus
              deref
                -40 (a6)
              28
            16
      4
  1
```

This code (in the appropriate format) is suitable as input for the code generator of our local C compiler. Feeding the output of the compiler through this code generator produces the assembly code given in figure 33.

Trivial optimizations, i.e. replacing the function call to multiply a value by 4 by a shift operation, are left to a peephole optimizer.

The steps in the code generation process for the second assignment statement

```
g (y). x (y) := 1;
```

are slightly different since *g* returns a value of type *asu_rec* instead of a value of type *au_rec*; *asu_rec* is an access type accessing objects of the constrained subtype *su_rec*; the descriptor for the value accessed by the return value of *g* is addressable, independent of the evaluation of *g*. The differences would have occurred when handling

```
Value (g (y). all)
```

instead of

```
Value (F (y). all)
```

and

```
Descriptor (g (y). all)
```

```
pea       10.w
pea       (a6)
movea.l   -32(a6),a0
jbsr      (a0)
addq.l    #8,sp
move.l    d0,-40(a6)
moveq     #28,d0
movea.l   -40(a6),a0
movea.l   (a0),a0
movea.l   #10,a1
suba.l    16(a0,d0.l),a1
pea       (a1)
pea       4.w
jbsr      lmul
addq.l    #8,sp
movea.l   -40(a6),a0
movea.l   (a0),a0
movea.l   #10,a1
suba.l    16(a0,d0.l),a1
pea       (a1)
pea       4.w
jbsr      lmul
addq.l    #8,sp
movea.l   -40(a6),a0
moveq     #1,d1
move.l    d1,4(a0,d0.l)
```

Figure 33. *Target assembler code for f (y). x (y) := 1*

instead of

Descriptor (f (y). all)

The code to be generated for

Descriptor (g (y). all)

reduces to code to address D_{su_rec}, the descriptor implementing *su_rec*. Since all values appearing in the type and subtype definitions are static, this descriptor is a global object.

5.4 Exceptions and exception handling

The subject of implementing exceptions seems interesting enough; several papers are devoted to a discussion about implementation strategies. Heliard [Heliard-83] discusses an implementation model. Baker et al [Baker-86c] discuss several implementation strategies

for exceptions and exception handling. Kruchten [Kruchten-86] spends a whole chapter on the subject.

According to chapter 11 of [LRM-83] exceptions are: "*errors or other exceptional situations that arise during program execution. ... To raise an exception is to abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen. Executing some actions, in response to the arising of an exception, is called handling the exception*". When an exception is *raised*, control is transferred to an *exception handler*. A handler is associated with the sequence of statements of a *frame*; a block statement, a subprogram body, a package body, a task body, or the main program. Selection of a handler for an exception depends on the dynamic nesting of units within the thread of control in which the exception is *raised*. The nesting of the units is determined by their calling sequence; not only by their static nesting. When an exception is *raised* in a construct without exception handler, the exception is propagated to the nearest enclosing context. If the immediately enclosing context is a task, the execution of the task is completed. If it is a library package or the main program, the execution of the main program is abandoned.

Exception handling and tasking interact in several ways. Tasking is supported by a tasking supervisor (chapter 6). This tasking supervisor is able to catch exceptions that are otherwise uncaught and is able to pass these exceptions on to other tasks if required.

A particular problem is how control is to be passed out of a master construct†. According to chapter 9 of [LRM-83] a master construct cannot be left until all dependent tasks are properly terminated (terminated in the sense of the Ada language). The compiler takes care of this problem by adding an additional exception handler to each master construct in the user's program. The code in this causes the thread of control suspended that is executing the handler until all dependent tasks have terminated. Consider the example of the program fragment

```
...
B: declare
   task type T is
     ...
   end;
   ...
   A, B, C : T;
   begin
     raise E;  -- some exception
   end;
   ...
```

Raising the exception *E* would cause the block *B* to be left. The block contains, however, three dependent tasks: *A*, *B*, and *C* that must terminate before leaving the block. The code

† A *master* construct is defined in chapter 9 of [LRM-83]; it is a language construct containing dependent tasks.

generated by the compiler is schematically:

```
   ...
B: declare
   task type T is
     ...
   end;
   ...
   A, B, C : T;
   begin
     raise E;     -- some exception
   exception      -- some compiler inserted handler
     when others => -- just catch any exception
          exit_master;   -- a call to the tasking supervisor
                         -- in which the thread of
                         -- control is suspended
        raise;   -- reraise the exception
   end;
   ...
```

exit_master is the name of the tasking supervisor procedure that takes care of suspending the current thread of control until a master can be left.

The handling of an exception in a rendezvous requires some special precautions to be taken. The supervisor takes care of all the actions that have to be performed at the end of a normal rendezvous, whether an exception was *raised* or not. (Such actions include the releasing of the calling task, they exclude passing of *out* parameters.) When an exception is raised within a rendezvous, the exception is propagated to both tasks that are involved in the rendezvous. From the caller's side it looks as though the exception was raised at the point of the entry call. In the callee the exception is propagated outside the accept statement. The tasking supervisor takes care of catching any uncaught exceptions and passing them on if necessary.

The tasking supervisor itself is also capable of *raising* exceptions. The circumstances under which a tasking supervisor must *raise* an exception are discussed in chapter 6. As an example, the tasking supervisor raises an exception when an entry call is made to the entry of a task that is either in the completed state or abnormal.

A simple and effective method to maintain information pertaining to an easy finding of an exception handler is by keeping a stack of *exception-handler-description blocks.* For any elaboration of an exception handler an exception-handler-description block is pushed onto the stack. An exception-handler-description block contains, apart from data identifying the context for the exception handler, a pointer to code that implements the exception handlers. For each thread of control a stack of exception handler description blocks is maintained.

On the exit from a construct in which an exception handler was elaborated, the exception-handler-description block is popped from the stack.

By applying this technique, the overhead of the run-time system remains limited. When an exception is *raised* the run-time support system follows the pointer to the exception-handler-description block in the current thread of control. It re-installs the correct environment and performs a jump to the address of a sequence of handler choices. No actions have to be performed for propagating handlers through constructs that do not have their own handler; the per-process pointer points to the handlers of the closest enclosing unit. The run-time cost for support of exceptions and exception handlers is caused by pushing handlers on the elaboration of a handler and popping handlers on the exit of their containing construct.

Within an exception handler several exception choices may be active. The dispatching of a choice is done by executing a sequence of *if-then-else* constructs. If, at the end of the sequence of exception choices, no match is found between the exception that was *raised* and the exceptions that could be handled in the choices, the exception is re*raised.* Consider the example of the construct

```
...
exception
  when X1 | X2 => S1;
  when X3    => S2;
  when others => S3;
end;
```

When a handler is entered, an identification of the exception which was raised is kept in a task-bound variable (called *current_exc*). The exception choice is searched for by executing a sequence of *if-then-else* constructs:

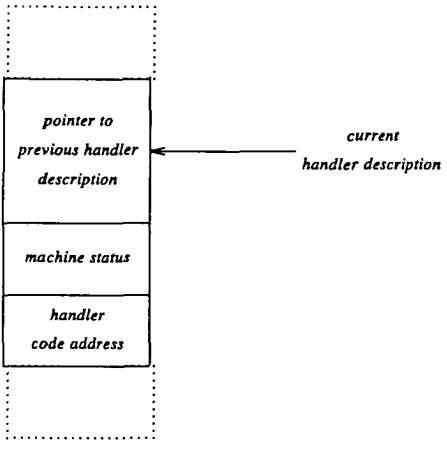
```
...
if current_exc = X1 or else current_exc = X2
then
  S1;
elsif current_exc = X3
then
  S2;
else
  S3;
end if;
```

This method is called *dynamic tracking* since the handler is tracked down dynamically. Baker et al [Baker-86c] also discuss an alternative method, a method in which a static mapping is applied to track down the exception choice that handles a particular exception. Apparently, the AIE compiler was designed to use this latter kind of handler track-down.

Exception-handler-description blocks can be maintained on the activation-record stack; no separate stack is required. For each elaboration of an exception handler a small

description block is pushed onto the activation-record stack in the current thread of control. The address of the topmost exception-handler-description block on this stack is kept in a location private to the enclosing task (the *current* handler).

In the Ada— implementation, an exception-handler-description block contains three components: a *link* to the previous exception-handler-description block, the target machine status at the point where the handler was elaborated, and a pointer to the actual handler code.



The link to the previous exception-handler-description block is needed to allow the previous handler to become the current handler again on the exit from a construct in which the handler was elaborated. The copy of the machine status is used to restore the context in which the handler was elaborated on processing an exception. Finally, the address of the handler code is required for obvious purposes.

On the normal exit from a construct in which a handler was elaborated, the exception-handler-description block is automatically popped from the stack. The handler found through the handler chain becomes the new current handler.

After the handling of an exception the current handler is popped from the stack. The handler, found through the description of the previous handler in the exception-handler-description block, becomes the new current handler.

The requirement to allow an exception to be re-*raised* after handling it in an exception handler, needs some special attention. Consider the following program fragment:

```
...
declare
```

```
...
begin
  ...
  raise A;
  ...
exception
  when A =>
    declare
      B : exception;
    begin
      raise A;
    exception
      when others => null;
    end;
    raise;
when others => null;
end;
```

The exception that must be propagated by the (re) *raise* statement is the one labelled *A*, not the one labelled *B*, although the exception labelled 'B' was *raised* most recently. The exception that is handled should be kept local to the handler rather than global. For each thread of control a *stack* of currently active exceptions is maintained.

Each exception must be identified by a unique identifier. In the Ada— implementation an exception is identified by the address of a memory location. To aid in debugging, this memory location has as its value the ASCII-encoded representation of the exception's identifier. This mapping scheme is possible since a linear address space is assumed. Kruchten [Kruchten-86] maps exceptions onto byte values.

5.5 The lowering of the semantics

In the expander phase a translation is made from a high-level intermediate program representation to a low-level intermediate program representation.

For the translation of a procedure, two scans are made over its intermediate tree representation. In the first scan the procedure's storage requirements are computed; in the second scan the low-level intermediate code is generated†. Storage allocation and selection of target-dependent operations are heavily influenced by the data-description model (earlier discussed in section 5.3) and the envisaged virtual-machine model (this virtual machine model is discussed below). The generated code does contain some target dependencies; the algorithms that are used in the expander to generate this code are highly target-independent, however. The implementation is parameterized to be able to deal with a variety of target-instruction-set architectures.

† Generating the low-level code may involve some more scans to be made over local expression trees.

In this section two major elements of the expander are discussed. First the envisaged virtual machine model is discussed, next the translation of a number of basic constructs is shown. The machine model is discussed in subsection 5.5.1, the generating of low-level intermediate code is discussed in subsection 5.5.2. In the latter subsection attention is given to the modelling of the elaboration of types and objects, the translation of names and expressions, in particular to function calling, constraint checking, aggregates, default initialization of record components and allocators. Further some attention is given to the translation of assignment statements and goto, exit, and return statements. Finally, the subject of elaboration code for (library) units is discussed.

5.5.1 Machine model

A number of assumptions on the structure of the real target machine is made. These assumptions are formulated in terms of some *virtual Ada target architecture*, to be discussed next.

It is assumed that the compiler is targeted towards a single-processor architecture with a linear address space. Data segments may contain global data, buffers and descriptor blocks for task activations. For each task object a task descriptor is allocated in a data segment; a task descriptor contains a region with space for activation records to be kept during the life-time of that task object. The amount of stack space that is allocated for a task object is determined by the compiler; no attempt is made to deal with stack overflow. More elaborate mechanisms, e.g. Berry heaps, and the consequences of their use are discussed by Gupta et al [Gupta-85].

Some effort has been made to keep the structure of the activation records compatible with the activation records usual for C [Johnson-81], in particular with respect to parameter transfer. An activation record in the Ada— implementation contains two parts: a statically-sized part (the so-called *fix-stack*) and a dynamic extension (the so-called *dyn-stack*). This structure is based on a classical description by Gries [Gries-71]. Fix-stack space is allocated on procedure entry; local objects of compile-time-determined size reside on this fix-stack part. Space for objects with a run-time determined size is allocated on elaboration of the declaration. Such allocation takes place by extending the *dyn-stack* part. A global register, the *base register*, points to the base address of the activation record of the currently active procedure. A second global register, the stack-top register, points to the first free location on the stack. Similarly to Gries' description, for each block (or similar structure) a *stack-top location* is maintained. In this location a copy of the stack-top register valid for that block is kept. On block exit the contents of the surrounding block's stack-top location are used to reset the stack-top register. Resetting the stack-top register implies discarding all dynamically allocated objects in the block being left.

Advantages of maintaining a stack-top location for each block are discussed by Gries: a stack-top location is useful when resetting the stack-top register on performing a goto or an exit from a loop. Furthermore it is advantageous for resetting the stack top after processing a composite value (the details of which are discussed later on.)

Addressing of a local object is relative to the base address of the current activation record. A local variable (and a local descriptor) is addressed with a negative offset; a

parameter is addressed with a positive offset. Addressing a non-local object is done by following the static chain until the base address of the activation record of the containing procedure activation is found and then adding an offset. Wegner [Wegner-68] describes an addressing scheme that uses a single central display. It is certainly possible to use a central display implementation.

The static link (the *environmental pointer*) for a procedure activation is passed as an additional parameter in the procedure call. A main level procedure, i.e. a procedure without an enclosing procedure, does not require a static link to its static environment since any object in its environment is addressable as a static object.

Space for parameters is shared between caller and callee as in the model described by Johnson et al [Johnson-81] for C. The caller pushes, in reversed order, the values of the actual parameters; the callee addresses the stack locations on which these values are kept through offsets relative to a base address. The caller is held responsible for throwing away the actual parameter values after the return from the function or procedure. Scalar *copy-out* parameters are copied back prior to adjusting the stack. Consider the example of the procedure fragment given below:

```
procedure p (a, b, c : in out integer) is
subtype my_array is array (1 .. b) of integer;
d, e : my_array;
begin
  ...
B1: declare
  f : my_array;
  begin
    ...
```

The layout of the activation record, after having elaborated the declaration of *f*, is given in figure 34.

The call/return model for functions with a *composite* return value requires some special attention. On return from a function with a composite return value, the activation record of the callee is preserved since it may contain the return value. The extent of the activation record therefore exceeds the extent of the call. To achieve this effect both function call and function return are considered composite actions. The action *returning* can be distinguished from the action of obliterating the activation record of the callee, similar to e.g. the model described for the SL5 language by Griswold et al [Griswold-77]. After having *used* the value, the caller resets the stack-top register. Since the result value has only a single *use*, the compiler is able to determine an upper bound of the value's extent. Notice that there is no interference with the copy-out mechanism for scalar *copy-out* parameters; functions just do not have *copy-out* parameters.

It is possible to maintain two registers instead of a single base pointer. One register, the *actual-parameter pointer* then points to the area on the stack where the parameters are kept, the other one, the ordinary base pointer, points to the base address of the activation record. In the Ada— implementation these two pointers collapse into a single base

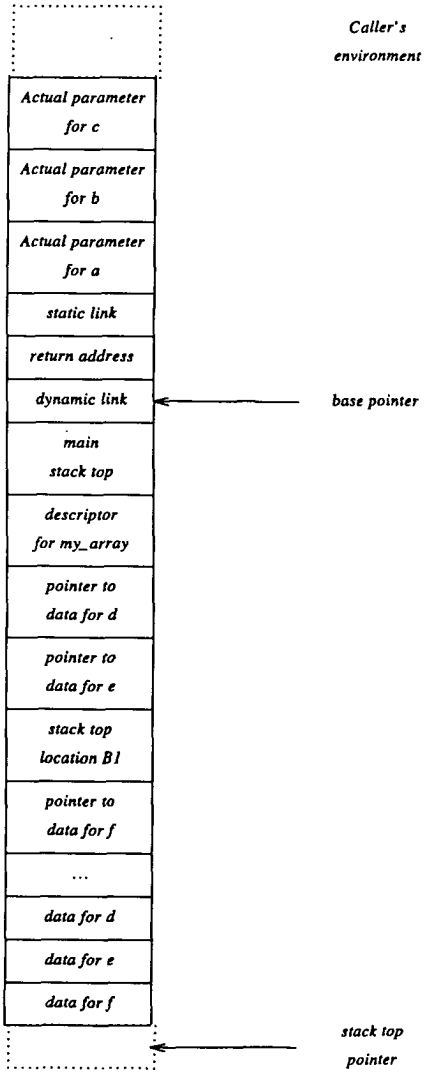


Figure 34. *Schematical overview activation record*

register. This choice is made because it is the model supported in the target C

environment. Even with the handling of parameters in a rendezvous†, no attempt is made to implement the pointers by different registers. This latter decision, however, may have to be reconsidered.

A task body is treated as an ordinary procedure with regard to the layout of its activation record. Any access from within a task body to a non-local variable is made as any other non-local access, through the static chain. A task body, however, is *never* called as a procedure; its activation is initialized by the tasking supervisor.

5.5.2 The generation of a low-level intermediate code

A main task of the expander is to generate a low-level intermediate program representation. In this subsection a brief survey is given of the structure of the code that is generated for various constructs. The enumeration is however far from exhaustive.

Types and objects

For each elaboration of a type, subtype or constraint, a descriptor will be constructed according to the data-description model. The expander generates descriptors, initialized as global data, or it generates code for the run-time construction of these descriptors. For simple static scalar types and subtypes, the compiler allocates global objects with static initialization. For such constructs, no run-time code is generated. For more complex descriptors, e.g. descriptors depending on dynamically-computed descriptor addresses or values, run-time code for the initialization are generated.

Often a check has to be performed on the legality of constraint values for subtypes and constraints. Although for simple cases this check could have been encoded in-line, a call to an appropriate run-time library function is generated. This decision was almost enforced by the code generator that is being used. This code generator takes expression trees as its input and, since the necessary checks often imply a multiple use of certain values, the code to perform the checks expands while the amount of temporaries also increases.

The elaboration of an object declaration proceeds in two separate steps: allocation and initialization. The amount of code to be generated for the allocation of an object depends on the storage class of the object. For global objects with a size that is known at compile time, no allocation code is generated: an assembly directive is produced to allocate at link-load time the required amount of storage. For fix-stack objects, space is reserved on the fix-stack part of the activation record; this space is allocated on procedure entry. For global objects with a size that is not known at compile time, the object is allocated on the dyn-stack part of the activation record of the main program. (The activation record of the main program does not have a fix-stack part, elements that logically belong to the fix-stack part of this activation record are implemented as globals.) The object is made addressable through a pointer in a global location; this pointer identifies the object. For a dyn-stack object the run-time stack is extended to accommodate space for the object. Space is allocated on elaboration of the object's declaration; the object is made addressable through a pointer kept in the fix-stack area.

---

† The actual parameters in an entry call are evaluated and stored on the caller's stack. The address of the actual parameters is passsed as a parameter in a rendezvous.

Initialization of one or more components of a record object, or the initialization of an access value, or the implicit creation of a task object, is always done by a run-time support function. A record type descriptor contains sufficient data for such a run-time routine to perform an iteration over the record components and to initialize these components.

Names and expressions

One of the most complex tasks of the expander is to generate efficient code for names and expressions. In the discussion of the implementation of the doublet model (section 5.3), the strategy applied to most of the simple forms of the name construct was discussed. We restrict ourselves here to some other cases, in particular to cases concerning calling sequences and parameter transfer, cases concerning constraint checks, cases on aggregates and cases concerning allocators.

Calling sequences and parameter transfer

Procedure-call mechanisms are very important; the time spent in calling a procedure, and in executing entry and exit code is often surprisingly long.

Code generators, such as the PCC code generator, provide a single operation to implement a function or procedure call. The user of the code generator does not have to worry about stacking parameters before the call and cleaning up the stack after the call. The underlying architectural mechanisms remain hidden from the user.

For each procedure call, the caller pushes onto the stack the values of the actual parameters, in reversed order, and then calls the procedure or function.

For an actual parameter corresponding to a scalar-formal parameter or an access-type-formal parameter only a value is pushed. For an actual parameter corresponding to a constrained-composite-formal parameter, a pointer to the actual-parameter value is pushed; the descriptor of the formal parameter is used to describe this value.

For an actual parameter corresponding to an unconstrained-composite-formal parameter, two pointers are pushed. The first one is a pointer to the value of the actual parameter, the second one a pointer to a descriptor for that value. (Pointers to descriptors are placed at the end of the actual-parameter list such that it remains possible to call C functions. These function may ignore the descriptor for any of the actual parameters.) The callee is able to address the stack locations through the names of the formal parameters; no value copying is done.

On return from the function, the caller is responsible for discarding the space occupied by the actual parameter values. An important advantage of sharing parameter space between caller and callee is simplicity of the implementation. Furthermore, it closely matches the calling mechanism in C.

Constraint checks

Constraint checks fall into three classes: range constraint checks, index constraint checks and discriminant constraint checks. Checks on range constraints are encoded in-line whenever possible.

Index constraints for one-dimensional arrays are implemented by in-line range-constraint checks. The checking of constraints in multi-dimensional arrays and the checking of discriminant constraints is done by run-time support functions. The implementation of checks for record-component selection was thoroughly discussed in subsection 5.3.2; such checks are implemented as range checks. The code generator uses expression trees in which such checks are not supported. Extra temporaries are used to store the values that must be checked.

Constraint checking on access types is similar to constraint checking on discriminants.

The *membership test* (the *in* operator) is similar to a constraint check. The difference is of course that the membership test generates a value, whereas a constraint check may *raise* cause an exception to be raised.

Aggregates

Any aggregate appearing in the text of an Ada program is made into an implicit procedure. For the elaboration of the aggregate, this function is called to yield the composite result value. In obvious cases the function is replaced by a table, containing an instance of the aggregate value.

A number of cases can be distinguished where the use of a complex mechanism such as a function with a composite return value is really an overshoot. In particular in those cases where an aggregate value can be constructed in the target location by simply performing a sequence of component assignments, a simpler implementation should be considered. The choices in the implementation model were influenced by the attempt to keep the implementation simple. Toetenel [Toet-84b] discusses some alternative forms for the implementation of aggregates in the Ada— implementation. He discusses forms of table-driven implementations and in-line encoded aggregates. It turns out, however, that for aggregates with complex elements, the method using a function is more elegant than other methods. In Ada+ a method is used similar to the in-line encoded forms described by Toetenel [Maddox-85]. Kruchten [Kruchten-86] discusses the evaluation of aggregate values by specifying a translation scheme for the various forms of an aggregate.

The structure of the function implementing the aggregate depends on:

- whether the aggregate is an array aggregate or a record aggregate;
- whether the array aggregate has an *others* choice or not.

The function implementing an aggregate is built up as a series of assignment statements of the component values to a temporary location within the function. The function result is the value of that location. In case of an aggregate occurring as aggregate element value, that value is also packed as a function or a table and evaluated on elaboration of the aggregate.

The front end normalizes the intermediate representation of record aggregates. The components are put in the order of the components of the corresponding record type while the *others* choice is eliminated. The function generated for a record aggregate consists of a sequence of assignment statements, one for each record component.

The expander dispatches array aggregates regarding the fact whether they have an *others* choice or not. For array aggregates without an *others* choice, the function implementing the aggregate is straightforward. It consists of a series of *for loops* and ordinary assignments as is demonstrated by the translation of the example below.

```
procedure test is
  type M is array (integer range <>) of integer;
  A : M (index_constraint) := (1 .. 10 => e1, 11 .. 100 => e2);
  ...
begin
    ...
end;
```

A function written in the Ada language that corresponds to the implementation of the aggregate in the example above is:

```
function anonymous return M is
  TEMP : M (1 .. 100);
begin

  for i in 1 .. 10 loop
    TEMP (i) := e1;
  end loop;

  for i in 11 .. 100 loop
    TEMP (i) := e2;
  end loop;

  return TEMP;
end;
```

An array aggregate that contains an *others* choice requires an index constraint from the context in which the aggregate is to be evaluated. Consider the example of a second program fragment below.

```
procedure test is
  type M is array (integer range <>) of integer;
  A : M (index_constraint) := (1 .. 10 => e1, others => e2);
  ...
begin
    ...
end;
```

A function in the Ada language that corresponds to the implementation of the aggregate in this example is:

```
function anonymous (X: M) return M is
  TEMP : M (X ' range);
begin

  for I in X' range loop
    if I in 1 .. 10
    then
      TEMP (I) := e1;
    else
      TEMP (I) := e2;
    end if;
  end loop;

  return TEMP;
end;
```

The index constraint, required to allocate a temporary object inside the function, is extracted from the actual target object on which the function is parameterized. The obvious alternative, viz. to generate the aggregate value straightforwardly in the target location, is not applicable according to the semantics of the Ada language: if a constraint error is raised during the elaboration of the aggregate, the target location should remain unmodified (section 5.2 4 of [LRM-83]).

Default initialization of record components

Another place, where expressions that appear in the program text are made into implicit procedures, is within records. For each initializing expression that appears in a record type definition the compiler generates a function that evaluates the expression when called. This function is activated by the run-time support system on the elaboration of a declaration of an object of the type (or an object containing a component of the type). The environment in which the expression must be evaluated is restored prior to the evaluation of the function. To that end, any record-type descriptor contains a copy of the base pointer of the activation in which the record type was elaborated.

Allocators

Storage for objects created by an allocator is allocated on a heap. The addressing issues of access objects at run time were discussed in subsection 3.4. of this chapter. Three different cases can be distinguished, as shown in the example below:

```
type M is array (integer range <>) of integer;

subtype SM is M (index_constraint);

type AM is access M;
type ASM is access SM;

A : ASM := new SM;      -- case 1
B : AM  := new SM:      -- case 2
C : AM  := new M (constraint);  -- case 3
```

1. The first case is exemplified by:

   ```
   A : ASM := new SM;
   ```

   In this case the object can be allocated in a straightforward manner. A descriptor for the object exists and only the object itself has to be allocated.

2. In the second case, exemplified by

   ```
   B : AM  := new SM;
   ```

   the descriptor for the object exists already; nevertheless, a full doublet must be constructed. The existing descriptor of the access object is copied onto the heap. The copying of the descriptor is required since there is no compile-time-known relation between the extent of the descriptor and the extent of the access object.

3. In the third case the descriptor for the allocated object must be constructed and the access object must be implemented as a doublet. An example of this case is

   ```
   C : AM := new M (constraint);
   ```

Statements

A large number of statements can be distinguished; only a few of them are discussed. The translating of most statement forms, e.g. conditionals, *case* statements and *for* statements is relatively straightforward.

Assignment statements

A value is implemented as a plain sequence of bits without any contained description. Therefore, assignment of a value can be implemented as a block-copy operation. Assignments in which a constraint check is required are performed by a run-time-library function. For those assignments in which a constraint check is superfluous, a distinction can be made between cases where the size of the data to be copied is known at compile time and cases where it is not. When the size of the data that must be copied is known at compile time, the compiler generates a block-copy operation for the assignment. The intermediate codes of both code generators, the code generator of a portable C compiler variant and the EM code generator, both provide a block-copy operation. When the size is not known at compile time, a run-time support function is called to perform the assignment.

The well-known problem of *overlapping slices*, e.g.

```
A (11 .. 20) := A (6 .. 15);
```

is dealt with by a run-time-library function. To avoid possible problems, any assignment involving a slice is implemented as a call to this run-time-library function.

Goto, exit, and return statements

Goto statements†, exit statements and return statements deserve some special attention. An implementation of goto or exit by a simple jump to the target location is not always possible because:

1. in case of an exit (and sometimes in case of a goto) the stack has to be unwound up to a point dictated by the target location;
2. possible master constructs have to be unwound;
3. possible exception handlers have to be de-activated.

For each of these problems a solution can be provided:

1. each textual region that determines the life-time of objects, has its own *stack-top location* (subsection 4.6.1); before the branch is made, the stack will be reset using the contents of the stack-top location of the region of the target location;
2. the number of statically enclosing master constructs for the above mentioned constructs is known at compile time. A call to a tasking-supervisor procedure is generated for each enclosing master to be left;
3. a similar situation arises with exception handlers for enclosing block structures. Since the nesting is a static one, all enclosing constructs for which an exception handler has to be deactivated are known and code can be generated to invalidate the exception handlers.

Consider the example of the construct:

```
declare
  task type T;

  task body T is ...
  ...
  end T;

begin
  ...
  for I in 1 .. 10
  loop
    declare
```

† The Ada— implementation currently does not support *goto* statements.

```
      A : T;
    begin
      exit;
    end;
  end loop;
end;
```

The exit statement appearing in the block enclosed by the *for* loop is suspended until task *A* is terminated. The compiler translates the program fragment above as if it looked like:

```
declare
  task type T;

  task body T is ...
  ...
  end T;

begin
  ...
  for I in 1 .. 10
  loop
    declare
      A : T;
    begin
      exit_master;   -- call to tasking supervisor procedure
                     -- that will cause the current thread
                     -- to be delayed.
      exit;
    end;
  end loop;
end;
```

A similar approach is taken in handling return statements.

The elaboration code for (library) units

Prior to the use of any declared object, its declaration must be elaborated. The elaboration of a declaration in an Ada language program may cause large amounts of code to be executed at run-time (by contrast to e.g. Pascal where the elaboration of a declaration is completely a compile-time event). Consider e.g. the declaration

```
A : array (expression .. expression) of integer;
```

On the elaboration of this declaration the subscript expressions are evaluated and a type descriptor has to be initialized. Furthermore, a subtype descriptor has to be initialized and storage for the object has to be allocated. Since this storage is allocated on the dyn-stack part of the run-time stack, the latter has to be extended and a location has to be initialized

with a pointer to the newly allocated space.

Packages, package bodies, procedure and function specifications and procedure and function declarations, all have to be elaborated before they or their constituents can be used. In particular, the body of a procedure or function should be elaborated prior to the use of the procedure or function. As discussed earlier (page 131), a compiler-generated local variable in the enclosing scope is associated to each function or procedure. On the elaboration of a function or procedure specification the entry address of an error routine is stored in that local variable. When the body of the corresponding function or procedure is elaborated the contents of that local variable are overwritten by the code address of the function or procedure. Any call to the function or procedure is made by using the contents of this local variable as an entry point. Therefore, when a function or procedure is called prior to the elaboration of its body, an error routine will be called.

The elaboration of a local package or a local package body does not require special actions; the elaboration code is just executed. The elaboration of library units, however, requires some care to be taken by the compiler.

The compiler equips each library unit with a small piece of entry code and exit code. At load time the compiler generates a sequence of subroutine calls to the library units, followed by a call to the procedure acting as main program.

The start-up code for a program with n library units to be elaborated is schematically:

```
start:  jsr   elabcode1
        jsr   elabcode2
        ...
        jsr   elabcoden
        jsr   main_program
```

The elaboration code of a library package or package body is embedded in function code. The function is highly artificial; the extent of its local data exceeds the activation of the function. The data allocated during the elaboration of a library unit may remain in existence during the lifetime of the program. The return of the embodying function is rather special, it is similar to the return of a function with a composite function result.

5.6 Intermediate codes and code generators

The main function of a low-level intermediate program representation is to provide a basis for retargetability. Retargetability is obtained by the isolation of target dependencies.

In the Ada— compiler a low-level intermediate program representation is used as the interface between the expander phase and a code generator. A careful design of the level of this intermediate representation made it possible to use an existing code generator for the final target-code generation. The code generator being used is the code generator belonging to the C compiler on our local system, a variant of the portable C compiler. As an experiment we have used EM code as the intermediate representation and the EM code generator as the code generator in the Ada— compiler (see also page 161). (This

experiment was performed when the compiler did not yet support tasking.)

In this section we discuss various code-generator issues. The contents of the section fall into 3 parts. In the first part we discuss the literature on low-level intermediate program representations. Next the intermediate code of a variant of the portable C compiler is discussed. Finally, we briefly discuss the experience with the adaption of the expander in the Ada— compiler to the EM code generator.

5.6.1 Literature

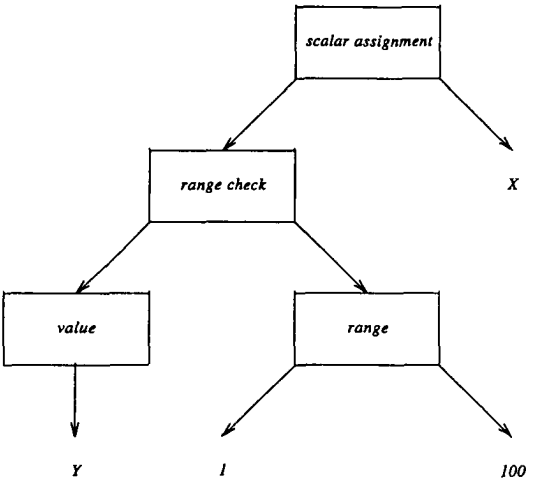
A number of papers on intermediate program representations can be mentioned.

The intermediate code of the European root compiler is discussed by Teller et al [Teller-81] and by Roubine et al [Roubine-82]. Teller et al discuss a number of design principles for a LLIL (Low Level Intermediate Language). They conclude that their intermediate language should be a tree-structured intermediate language. The LLIL reflects the syntactic structure of the source program, allowing for an easy and systematic generation of the LLIL representation from a high-level intermediate representation.

Roubine et al [Roubine-82] discuss the design principles of the resulting low-level intermediate language Lolita, (*Low Level In*Termediate L*anguage*). It is a tree-structured intermediate language, bearing some resemblance to the intermediate code of the portable C compiler (to be discussed later). However, since Lolita has been designed for the Ada language, it supports a number of features specific to that language. Consider the example of the assignment statement

```
X := Y;
```

The intermediate representation is given below; in this representation *X* has a static constraint, the integer subrange 1 .. 100.



Appelbe et al [Appelbe-82] discuss an operational definition of an intermediate code for a portable compiler (the Telesoft compiler). Their approach is to define an intermediate I-code and an so-called A0 architecture. The I-code is evolved from P-code, a compact code for a simple stack-oriented pseudo-machine for Pascal. I-code includes operators for stack expression evaluation, load and store operations for local and external data segments, subprogram calls and branch instructions and calls to a run-time support package. The main goals in the design of the A0 architecture were simplicity, completeness and compatibility with existing instruction set architectures. It must be possible to directly map I-code to A0 instructions and to simulate the behaviour of the A0 machine on a wide variety of actual machines.

Yet another design of a low-level intermediate code was given by Ibsen et al [Ibsen-82]. A virtual machine, the A-machine, was designed as part of the PAPS project. The main objectives in the machine design were:

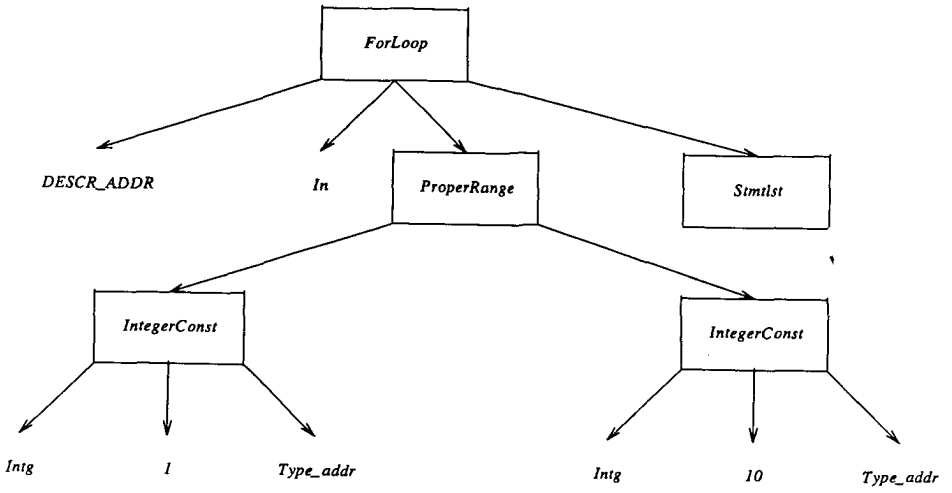
- to construct an instruction set which allows an efficient representation and execution of Ada programs;
- to define a level of portability by making a design which is implementable on existing hardware.

The specification of the machine is complete; the formats of instructions, data units, descriptors etc. have been specified. The A-code is generated by the back end of the DDC Ada compiler. It is generated from a tree-structured intermediate code, IML_7, in which most of the structural information of the original Ada program remains. Consider the

example of a *for*-statement in Ada:

```
for I in 1 .. 10
loop
   statement list
end loop;
```

The structure in IML_7 (source: [Clemmensen-83]) is:



The resulting A-code is (in a symbolic form):

```
      PushConstant     1
      PushConstant     10
      PushAddr         I'Address
      StartLoop        L2
L1:   Statement list
      EndLoop          L1
L2:
```

Finally, it is worth mentioning that the BreadBoard compiler [Nowitz-82] and the Irvine compiler [Irvine-83] both use the programming language C as intermediate language.

The Karlsruhe Ada compiler employs a low-level intermediate language called AIM, *A*bstract *I*ntermediate *M*achine. The AIM serves as the target machine for those parts of an Ada compiler that do not strongly depend on the target machine. AIM programs

consist of several parts, the expression parts are tree-structured. The intermediate language further contains other operators and directives.

5.6.2 The portable C compiler

The portable C compiler, PCC for short, was designed and implemented by Johnson [Johnson-78]. A main goal in its design was to achieve a high degree of retargetability. The approach in obtaining this retargetability was by isolating target dependencies in the code generator's implementation. This approach, advocated by Johnson, significantly deviated from the usual approach in which a hypothetical intermediate-instruction-set architecture is used. An example of a hypothetical intermediate-instruction set is a.o. P-code for Pascal [Nori-74]. The basic assumption in Johnson's approach is that a code generator should be parameterized on the semantics of the target machine. Target dependencies are isolated in a few routines and a table driving the code generator. These dependencies account for about 25 % of the source of the code generator.

Although it is possible to configure a two-pass version of the compiler, it is common to have it as a single-pass compiler. In the two-pass version the compiler front end translates C source programs into low-level expression trees, intermixed with target assembly code for inherently target dependent constructs. The code generator, the compiler's second pass, then reads the intermediate programs, passes on each assembly line read, and translates the low-level expression trees into the target-assembly code.

The PCC has been retargeted towards a variety of architectures, including a DEC PDP-10 instruction-set architecture, the M68000 family and Risc instruction-set architectures. Although the compiler was designed for the C programming language, the code generator turned out to be quite useful as a code generator for other language processors as well. Well-known examples are the UNIX F77 compiler [Feldman-79a] and the Berkeley UNIX Pascal compiler [Joy-83].

A large number of versions and variants of the PCC is available, in particular under UNIX systems. Source versions for various targets are available on UNIX distribution tapes. In the Netherlands a software house (ACE) has developed its own version of the PCC. In this version most target dependencies of the intermediate code are eliminated. The code generator of this version is used in conjunction with front ends for C, F77, Pascal, Modula 2 and the Ada— compiler. It is targeted to VAX architectures, the M68000 family, the ND500 and the NS16000 family. Since the UNIX system on our hardware was delivered by ACE, the choice of their compiler back end was a rather natural one. Their compiler will be referred to as ACE-PCC, their intermediate code as the ACE-PCC intermediate code.

An ACE-PCC intermediate code program consists of low-level expression trees and target-relative directives.

The expression trees are given in some prefix form. The operators for the intermediate tree are derived from the set of operators in the C programming language. For almost each operator in the C language, there is an operator in the intermediate representation. In addition to the usual arithmetical operators, an *assign* operator, a *call* operator and a *dereferencing* operator are supported. A detailed description of the intermediate code is

given in [ACE-85].

The code generator manages the target register set on a procedure by procedure base. The intermediate code provides directives to indicate the textual start and the textual ending of a function or procedure. C does not allow static procedure nesting and consequently the code generator cannot handle static procedure nesting. Compiler phases operating prior to the code generator, have to reorder the intermediate representation.

The code generator generates a C-style function return on encountering an end-of-procedure-text directive. Consider the example of a simple C program given below:

```
main ()
{
   int a, b, c;

   a = a + b * c;
   return a;
}
```

The intermediate code, that is generated for an M68000 based system, is given below. Notice that, since the semantic level of the intermediate-code operations is low, the intermediate-code size expands very rapidly.

```
p_int is used as abbreviation for pointer to int
ICLANGUAGE    C
                ...
ICGLOBL         _main
ICFBEG          1       12   0    _main

[1        128     7      13
.5                                    -- expression starts with ". line number"
assign    (int)
deref     0       (int)
subtract  (p_int)
register  0       R14    (p_int)
const     8       0      (int)        -- 8(r14) = a

add       (int)
deref     0       (int)
subtract  (p_int)
register  0       R14    (p_int)
const     8       0      (int)        -- 4(r14) = a
mul       (int)
deref     0       (int)
subtract  (p_int)
register  0       r14    (p_int)
```

```
const     12      0       (int)      -- 12(r14) = b
deref     0       (int)
subtract  (p_int)
register  0       r14     (p_int)
const     16      0       (int)      -- 16(r14) = c

.6
force     0       (int)              -- compute expression in r0
deref     0       (int)
subtract  (p_int)
register  0       r14     (p_int)
const     8       0       (int)

          ...
]
ICDATA
```

Using the ACE-PCC low-level intermediate code as output for the expander poses only a few problems. The problems occur in those areas where the semantics of constructs in the Ada language cannot be mapped directly onto constructs in the C language. The main problem areas are:

- procedure nesting. The C programming language does not (yet) support statically nested procedures or functions. The expander of the Ada— compiler has to unravel static procedure nesting to allow the use of the ACE-PCC code generator;
- function and procedure calling. The ACE-PCC intermediate code contains a call operator to support function calls. The binary variant of this operator takes two parameters: an expression yielding the function entry point and a *list* of actual parameter expressions. The *call* operators implement the C semantics, i.e. the passing of the actual parameter values to the called function or procedure, the calling of this procedure, and the cleaning up of the space occupied by the parameter values.

  Calls of procedures with *copy-out* parameters or calls of function with composite return values can not be mapped onto the binary C calling operator. Copying-out of scalar parameters is done by code explicitly generated by the compiler's expander. Similarly, code to handle composite return values is also explicitly generated by the expander.

Consider the example of the following Ada procedure and its translation into the high-level intermediate representation, the low-level intermediate representation and M68000 target assembly code.

```
procedure EXAMPLE is
a : array (1 .. 10) of float;
begin
  a (3) := 1.0;
end;
```

The high-level intermediate representation (in ASCII, omitting needless details):

```
Unit = 728
1:  [WITH EXAMPLE]
2:  [SUBBODY EXAMPLE
      specif => (726, 0)
3:  [ARRAYTYPE %50AAB   -- anonymous arraytype
     elemtype => (1, 14)
     [INDEX
       indextype => (1, 4)]]
5:  [SUBTYPE %50AAC     -- anonymous subtype
4:   [CONSTRAINT
       on  => (728, 3)
       [INDEXCONSTRAINT
         [RANGE
           rangetype => (1, 4)
           [EXPR
             exptype => (1, 4)
             [LITERAL
               literaltype => (1, 4)
               value => "1"]]
           [EXPR
            exptype => (1, 4)
            [LITERAL
              literaltype => (1, 4)
              value => "10"]]]]]]
7:  [OBJECT "A"
6:   [OBJDESC
       objtype => (728, 5)]]
   [ASSIGN
     [INDEXING
       type => (1, 14)
       [EXPR
         exptype => (1, 4)
         [LITERAL
           literaltype => (1, 4)
           value => "3"]]
       [NAME
         type => (728, 5)
         fentity => (728, 7)]]
     [EXPR
       exptype => (1, 4)
       [LITERAL
         literaltype => (1, 14)
         value  => "1.00000"]]]
```

The low-level intermediate code (omitting superfluous details) becomes:

```
ICBSS
ICLABN 3
DOUBLE 1.00000e+ 00
ICTEXT
ICGLOBL MalHaaaE  (internal name EXAMPLE)
ICFBEGS 4 6 0 MalHaaaE
[4 768 7 13 MalHaaaE
 ....
assign (float)
deref 0 (float)
add   (pointer to float)
subtract (pointer to float)
register 0 r14 (pointer to int)
const 96 0 (int)
mul   (long int)
const  8  0 (int)
const  3  0 (int)
name L3 (float)
]
ICDATA
ICGLOBL _S726  --start label
ICLABS  _S726
```

The generated M68000 assembly code is given below:

```
 .data
 .L3        .double 1.0000e+ 00
allaac50    array descriptor
            4
            1
            1
            _mach_real
            _integer

 .data
allaae50    array-subtype descriptor
            5
            allaac50
            0
            4
            1
            10
```

```
.text
        .globl MalHaaaE
MalHaaaE
        link        a6, #-F4
        movem.l     #S4, -F4(a6)
        move.l      sp, -8(a6)
        moveq       #3, d0
        sub.l       #1, d0
        move.l      d0, -(sp)
        pea         8.w
        jbsr        lmul
        addq,l      #8, sp
        add.l       a0, d0
        subi.l      #96, d0
        movea.l     d0, a0
        move.l      L3, (a0)
        move.l      .L3+4, 4(a0)
        unlk        a6
        rts
        .globl      _S726
_S726   .long       MalHaaaE
```

Experiences with another code generator

An early version of the compiler back end was changed to generate EM code as the low-level intermediate program representation. The experiment is discussed by Barkey et al [Barkey-85]. The compiler was at that time incomplete. Features as e.g. tasking and generics were not yet implemented.

EM is a family of intermediate languages, designed by Tanenbaum et al [Tanenbaum-83a]. EM is available as part of a *compiler writer's toolkit*, the Ack (Amsterdam compiler kit) tool kit [Tanenbaum-83b]. It is designed to support a variety of languages. Tanenbaum et al [Tanenbaum-83b] state that "... *Typical languages that could be supported include Ada, ALGOL 68, BASIC, C, FORTRAN, Modula, Pascal, PL/I, PL/M, PLAIN and RATFOR, whereas Cobol, LISP and SNOBOL would be less efficient.*". Languages for which compilers to EM are available are C, Pascal, BASIC, PLAIN and Modula 2.

To form an opinion on the quality of the code generators, several programs were translated and their output was measured. It must be noticed that no optimizer other than the more or less standard EM peep-hole optimizer was used. Barkey et al [Barkey-85] discuss the comparison in detail; the main observation is that the differences in the code sizes are quite small. In the table below code sizes are given for some example programs.

Code size comparison			
program	*EM code*	*ACE-PCC code*	*source lines*
p1	*544*	*548*	*60*
p2	*2552*	*2508*	*210*
p3	*8096*	*7876*	*337*
p4	*4448*	*4096*	*445*
p5	*20224*	*20380*	*1500*

Code sizes for some example programs

The differences in the code sizes were less than expected. The largest difference is less than 10%. By carefully examining the generated code it became apparent that improving the quality of the EM code was merely a matter of adding patterns to the code-generator description.

6. The Ada— tasking supervisor

The Ada programming language allows certain program units, tasks, to operate in parallel and to communicate with each other during their execution. Task communication in the Ada language is based on CSP, Communicating Sequential Processes [Hoare-78], a model well known in the literature. Tasking is a subject appealing to people's imagination, the notion of parallellism apparently attracts many people. The last few years a tremendous number of papers was published on applications of Ada tasking. Just on the topic of the so-called dining philosophers several papers appeared.

The implementation of tasking also received a fair amount of attention; a large number of complete or partial implementation proposals has been published. Most authors follow an approach in which a *tasking supervisor* is defined. This tasking supervisor then takes care of the non-sequential actions. Given a well-designed interface to such a tasking supervisor, the code of a compiled Ada program is essentially† equivalent to the code that is generated for a sequential Ada program with explicit calls to tasking supervisor services.

In contrast to other simpler languages, e.g. Modula II, in-line encoding of task management functions and task communication functions is not sensible. The amount of code involved is far too large. Therefore task management and task communication are supported by a tasking supervisor. A simple and clean supervisor interface for Ada tasking is desirable for several reasons: it is important that customized run-time systems can be built without any knowledge of the compiler structure and furthermore, given a simple consistent interface, the work of the compiler writer is much easier. It is certainly possible to define and to implement a tasking supervisor for the Ada language semantics that is implemented as a library package or even in firmware.

A potential drawback of a clean and simple supervisor interface for Ada tasking is the overhead incurred in the tasking supervisor. Any extra layer of software causes overhead, decreasing the real-time capabilities of the implementation.

This chapter contains a survey of the literature and a discussion on a tasking supervisor for the Ada— compiler. The contents of the chapter fall into 4 parts. The chapter starts with a brief review of the literature; next, general tasking issues and requirements for an Ada tasking supervisor are discussed. Then a schematic overview of our own tasking-supervisor interface is given and some implementation issues are discussed.

An implicit assumption throughout this chapter is that the tasking supervisor must be implemented on a *single-processor* system. Attempts to implement partially or completely the tasking semantics of the Ada language on multi-processor systems are discussed in the literature [Ardo-84], [Cornhill-83], [Cornhill-84], [Jones-82], [Rosenblum-87], [Weatherly-84], [Wellings-84]. From our point of view this remains a topic for future research. The current Ada— tasking kernel runs on an M68000-based system under the

† The reader is warned that it remains in general impossible to write a sequential Ada program with calls to a supervisor interface that behaves like a tasking Ada program. Various supervisor procedures deal with addresses of code. The manipulation with code addresses in Ada programs is only possible using machine-dependent features.

UNIX operating system. We are convinced that essentially the same supervisor interface can be used on closely-coupled multi-processor systems; there is as yet, however, no hard evidence either to prove or to disprove this claim.

Although a fair amount of literature on tasking supervisors is available, there is hardly any description of tasking supervisors for commercially-available Ada compilers. It is a pity, though understandable from a commercial point of view, that such information is not available.

As a final introductory remark, note that we do not pretend to evaluate the tasking facilities of the Ada language itself. The reader interested in an evaluation of these facilities is referred to e.g. Roberts et al [Roberts-81] for a discussion on the use of Ada on multi-processors, to Burns et al [Burns-85] for a discussion on Ada tasking in general or to Bayan et al [Bayan-86b] for a discussion of requirements for Ada tasking supervisors. Burns et al present a review of the whole Ada tasking model.

6.1 A survey of the literature

The very first description of the tasking model for preliminary Ada and its implementation is given in the Ada rationale [Rationale-79]. The implementation model for rendezvous is based on message passing. Tai [Tai-80] points out errors and omissions in the suggested implementation.

Haridi et al [Haridi-81] report on an early implementation of the real-time primitives for the Ada programming language. The paper is an extended abstract of an earlier report [Bauner-80]. As part of a masters thesis project, a large part of the tasking primitives of *preliminary* Ada [Ada-79] was implemented on a PDP-11. The goal of the study was, apart from obtaining a run-time supervisor, to compare the Ada tasking mechanism with the tasking mechanism in CS. CS is, according to Bauner, "*a language extended from C ... based on the real-time kernel SIMON*". The primitives of the tasking supervisor were written in PDP-11 assembler code; they are callable as ordinary C functions. The authors themselves are quite positive about their supervisor; they compared their straightforward implementation of the Ada tasking primitives with the implementation of a set of primitives for the CS language. Experiments indicated that the performance of both kernels was similar. It must be realized, though, that it was never the intention of the authors to provide a system for use with a real Ada language compiler; only a subset of the Ada language tasking semantics was implemented. In particular the notions *master* and *dependency on masters* and the handling of exceptions remained unimplemented. The size of the whole supervisor is about 2,000 words of code and data.

Dijkstra et al [Dijkstra-83] discuss an attempt to adapt Bauner's supervisor to the semantics of the revised Ada language. As part of a student assignment, they implemented a version of the tasking supervisor on and for the PDP-11 in assembly code. They reported that the size of their still incomplete supervisor was about 3,600 words of code and data. Later, this supervisor was rewritten in C and ported towards an M68000-based system.

Nassi et al [Nassi-80] discuss a well-known optimized form for the implementation of task bodies. Whenever a task body consists of a loop with a selective-wait construct it is possible to define a translation in which the task body has no own process associated with it.

Stevenson [Stevenson-80] discusses algorithms for translating Ada multitasking constructs into multitasking constructs for the Ada-M† language. At least at that time the study was a paper study. According to Stevenson: "*The purpose of the translation is to study various implementations of Ada tasking and their relative problems, merits, and efficiencies*". It was shown by examples that a number of Ada constructs could be mapped onto constructs in Ada-M. Furthermore, Stevenson discusses various implementation strategies for the Ada rendezvous mechanism. The discussion in subsection 6.2.3 is based on it.

Stevenson's ideas have formed the basis for a supervisor presented by Falis [Falis-82]. The supervisor of the latter also uses Ada-M as a target language. One of the merits of the discussion by Falis is that the problems that must be dealt with in the design of an Ada tasking supervisor interface are described systematically. According to Falis: "... *The facilities provided should allow implementation of tasking using any of the message passing, procedure call or order of arrival transformation strategies and should support implementation of the full tasking semantics. ... Our assertion that this interface is sufficient is based on testing done to this date on a supervisor we have implemented at Stanford...*". The supervisor interface was presented as a single package, the specification of the semantics is incomplete, however. It remains hard to imagine, even after having studied Stevenson's work, what the precise semantics of the various procedures are.

Rosen [Rosen-83] discusses yet another Ada tasking supervisor. He emphasizes rendezvous management; neither task creation nor task activation are discussed. The work is interesting though for two reasons. First, much attention is given to a reduction of the supervisor interface's size, as well as to a reduction of its complexity. Second, a complete prototype implementation description of the supervisor is given in terms of Ada packages. It is rather obvious, just by inspecting the code, that the Ada program texts have never been fed through a compiler. Nevertheless, we took definite advantage of its availability in the design and in the implementation of our own supervisor. The size and the complexity of the interface have both been reduced by the unification of all kinds of entry calls and all kinds of accept and selective-wait statements. Selective-wait constructs are implemented by a single supervisor procedure; similarly all kinds of entry call (ordinary, selective or timed) are implemented by a single supervisor procedure (see section 3 of this chapter).

Recently, Rosen presented a doctoral thesis [Rosen-86] (in French) in which he discusses some elements of a run-time system for Ada. A chapter in his thesis is devoted to a discussion on the implementation of tasking primitives in the virtual Ada machine (which was discussed by Kruchten [Kruchten-86]). Essentially the same interface as was described in [Rosen-83] for the handling of task management is given in [Rosen-86].

---

† Ada-M is a local variant of Ada with lower-level tasking primitives developed and in use at Stanford University (California, USA).

Kamrad [Kamrad-83] discusses an overview of the run-time environment for Ada tasking programs in the Ada Language System environment. The paper emphasizes run-time storage management structures rather than the architectural aspects of a supervisor interface.

Leathrum [Leathrum-84] also discusses a run-time tasking supervisor for the support of Ada task management. The interface to the supervisor is presented as an Ada package and the Ada text of a procedure implementing the supervisor activities of a rendezvous is given.

Baker and Riccardi [Baker-84], [Riccardi-85] present a rather detailed description of the run-time tasking supervisor for their FSU Ada cross-compiler, targeted for the Z-8000. The description of the supervisor is presented in the form of Ada procedures and functions; unfortunately, the data-structure definitions are not included. The description is given in two parts. In the first part [Baker-84] elements of task management are discussed. As a follow-up in the second part [Riccardi-85] rendezvous management is discussed. Finally, a rationale for the design of an improved version of the supervisor is given in [Baker-86a].

The interface proposed in [Baker-86a] does not only address issues as establishing critical regions, setting time slices and handling dynamic priorities, it also addresses binding interrupts to accept statements and the implementation of a tasking supervisor on a multi-processor system. As such it goes far beyond the requirements as dictated by the language definition. A more general outline of tasking and the consequences of the implementation of supervisors is presented in [Baker-86b]. Finally, in [Baker-86c] attention is paid to the implementation of exceptions, in particular in the context of a tasking environment.

Persch et al [Persch-84] and Bayan et al [Bayan-86a] report on a portable Ada tasking supervisor designed for the Karlsruhe Ada compiler. The supervisor is well-defined and well-documented. One of the interesting points in the description is that translation schemes are presented that indicate what the structure of translated tasking constructs is. The use of the various supervisor procedures and functions is illustrated by a variety of examples in which (pseudo-)translated forms of task bodies, selective-wait constructs and other constructs are shown.

Within the context of the project *Asterix*, sponsored by the Multi Annual Programme of the commission of the European Communities, a study on requirements for an Ada tasking Supervisor was done in 1985/1986. Bayan et al [Bayan-86a] report on that study. The goal of the Asterix (*A*da *S*ys*T*em for *E*mbedded *R*eal-time *I*ndustrial e*X*ecution) kernel is to provide a basis for the use of the Ada language in industrial real-time applications on micro-computer systems. The Asterix kernel must be seen as a standard kernel onto which tasking-supervisor interfaces for Ada compilers can be mapped. The proposed implementation is based on a two-level layered approach. A lower level - perhaps even implemented in firmware - provides the primitives for the upper level; it is this lower level that must be regarded as a potential standard. The upper level is a tasking supervisor interface interfacing to a particular compiler. It can be implemented in terms of the lower

level. This upper level depends on characteristics of the compiler.

The Asterix kernel is not only meant as the supervisor for tasking, it also provides an interface for a tracer and an Ada spy. Bayan et al have used the Karlsruhe Ada tasking supervisor to demonstrate the viability of the Asterix kernel. It remains an open question what price, in terms of efficiency, must be paid for such a layered approach. If a certain performance loss is acceptable, however, it may be worthwhile to have a standardized interface on which tasking supervisor implementations for specific compilers can be developed.

Zang [Zang-86] proposes yet another scheme for the implementation of communication and synchronization mechanisms of Ada. The proposal is based on a minimum operating system supervisor. Primitives and data structures that are used to interpret the concurrent Ada language constructs are discussed. Furthermore, mapping schemes are given to translate Ada's concurrent statements to calls to the supervisor primitives.

Standardization of a tasking supervisor interface is so important that a special working group of SigAda (*ArteWG*) was established to look into standardization problems. The goals of the working group are to identify target dependencies in run-time support systems and to define portability guidelines. A preliminary paper [SigAda-86] was published.

Hilfinger [Hilfinger-82] gives an overview of forms of tasking idioms for which specialized translation forms are possible. Hilfinger recognizes that there is a general fear of the price to be paid for the large overhead which is expected in using a large number of tasks. The use of a large number of tasks probably causes suffocation of the system. The main contribution of Hilfinger is that cases are identified where tasks can be executed without having an own process. What Hilfinger essentially describes is the design of a set of very specialized schedulers, suitable for subsets of the set of tasks comprising a program.

Jones et al [Jones-82] report on a study on the comparative efficiency of different implementations of the rendezvous mechanism of the Ada language. The set-up is, although very interesting, not applicable to our situation. Their exercise was done on the C_m^* system, a multi-processor system developed at Carnegie Mellon University (Pittsburgh, U.S.A.). The conclusion of the study was that for more than about 4 LSI-11's a hand coded implementation, using a busy wait on shared variables, was superior to other implementation strategies. At a later stage Ardo [Ardo-84] continued the discussion. He provided a description of an experimental implementation of an Ada run-time system on the multi-processor C_m^* system.

Weatherly [Weatherly-84] describes a proposed network operating system to support distributed Ada tasking. The proposed supervisor would operate based on message passing. In particular message handling that is oriented towards task control is interesting; it clearly indicates the complexity of the implementation of task completion and termination.

A number of papers discusses formal description techniques for Ada tasking. Most of these papers, e.g. [Belz-80], [Clemmensen-82] and [Bondeli-83], are related to a

description of the semantics of tasking without having to rely too much on a particular implementation strategy. The use of such descriptions is primarily found in supporting the process of modeling an implementation.

Finally, some papers express concern on the efficiency of the implementations of tasking in the Ada language. Burns et al [Burns-85] report that the time spent to accomplish an empty rendezvous (i.e. a rendezvous without parameters and with no code in the accept body) takes about 1 msec using the York compiler. Baker [Baker-86a] makes similar remarks on his implementation. Burger et al [Burger-87] report on the overhead associated with tasking facilities in the Ada language. They report that task activation and termination takes almost 2 msec on a VAX 8600 using the DEC compiler. The execution of a rendezvous took about 0.5 msec. Our experience is similar. We found that the average time in which a rendezvous was performed, was almost 1 msec on a M86020 based system under UNIX. Huijsman et al [Huijsman-87] discuss in more detail the relation between the overhead and particular implementation strategies for tasking in the Ada language.

6.2 Issues in an Ada tasking supervisor

In this section issues that must be dealt with in the design and implementation of an Ada tasking supervisor and its interface are discussed. The section heavily relies on chapter 9 of [LRM-83].

Care must be taken not to overspecify the supervisor interface and to minimize constraints which are put on the user of the interface, i.e. on the compiler. Following Falis [Falis-82], we do *not* deal with:

1. algorithms for deciding whether a rendezvous may be scheduled immediately and what thread of control should execute the rendezvous code;
2. entry-parameter handling. It is assumed that the compiler can generate code to address the actual parameters of an entry call in a rendezvous and can address the variables in the static environment of the accept statement in which the rendezvous takes place;
3. optimizations of specific task forms [Nassi-80], [Hilfinger-82];
4. storage allocation issues. It is assumed that a complete run-time support system of which the supervisor is a part, deals with such issues.

Facilities dealt with, which *are* required by the Ada tasking semantics can be divided into three parts:

1. book-keeping;
2. management of tasks;
3. communication of tasks.

These issues will be discussed in some more detail.

6.2.1 Book-keeping

Any supervisor should support some general forms of book-keeping, since e.g. the length of the queue of elements waiting for an entry *E* must be determinable through a query with the *count* attribute (i.e. *E' count*). The decision as to whether a rendezvous may begin when an accept statement is encountered depends on the state of the queue, as does the decision as to whether an open accept alternative may be selected (see also sections 9.5 and 9.7 of [LRM-83]).

According to 9.4 of [LRM-83] each task *directly* depends on a single master. A *master* is a construct that is either a task, a currently executing block statement or subprogram or a library package. The dependence on a master is a *direct* dependence in the following cases:

1. the task designated by a task object that is the object or a subcomponent of the object created by the evaluation of an allocator depends on the master that elaborated the corresponding access-type definition;
2. the task designated by any other task object depends on the master whose execution created the task object.

It seems useful to distinguish between masters that *do* have dependent tasks and masters that do not have such tasks. Baker [Baker-86a] introduces the term *trivial masters* for masters of the latter kind.

A tasking supervisor should for a number of reasons have knowledge about task objects and the masters they depend on:

1. a thread of control must be prevented from leaving a master until all tasks dependent on that master are terminated;
2. when a task is in the argument list of an abort statement and the abort statement is executed, then not only the task itself must be aborted, but so must all its (recursively) dependent tasks;
3. an open *terminate* alternative within a selective-wait construct can only be selected when the master and the dependent tasks fulfil certain conditions.

Any supervisor should maintain a representation of dependencies between the various masters and their tasks, as well as the various scopes being executed by tasks and possibly having dependent tasks themselves. An implication is that a supervisor should be notified on entry to and on exit from any non-trivial master.

The functional behaviour of an Ada tasking supervisor can be modelled as a state diagram. Bondeli [Bondeli-83], Hartig [Hartig-81] and Bayan et al [Bayan-86a] provide examples. Two partial state diagrams are presented for the discussion on tasking semantics. First, a state diagram is presented that describes the *task-management* issues. Second, a state diagram is presented with which *task-communication* can be modelled. It must be realized, however, that transitions can take place between states in the different diagrams; such transitions are *not* indicated here.

6.2.2 Management of tasks

A first point of discussion is the management of tasks. From the point of view of task management a number of *states* can be distinguished. These states are first mentioned and then explained below.

1. created;
2. activating;
3. activating subtasks;
4. executing;
5. at end of master;
6. completed, waiting for termination;
7. terminated.

A complete state diagram is given in figure 35.

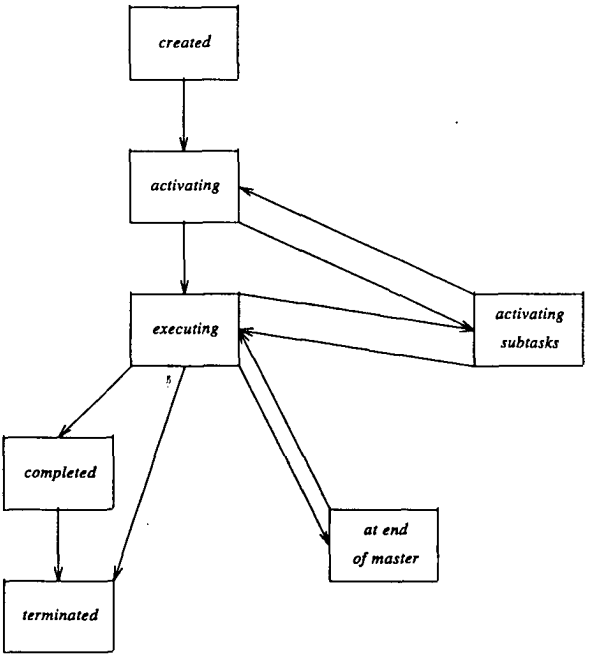


Figure 35. *State diagram for task management*

The states appearing in this diagram are now discussed in more detail.

1. A task is said to be *created* on the elaboration of the object declaration declaring the task object. The creation of a task must be distinguished from its activation. According to section 9.3 of [LRM], the activation of a task created by an object declaration does not start prior to completion of the elaboration of the declarative part in which the object declaration occurred. The creation of a task entails the sending of initial context information to the supervisor. It also includes sending the identification of the master on which the task depends to the supervisor. Furthermore, some identification of the task, viz. its *name*, must be made known to the supervisor so that it may be referred to within other supervisor requests.

2. The initial part of the execution of a task, i.e. the elaboration of the task body's declarative part, is called the *activation* of the task object and also that of the designated task. During this process, subtasks can be created. After the elaboration of the declarative part, but prior to the execution of any statement, these subtasks should be activated. Each task in the set of subtasks is either made *executing* or it is in the completed state, i.e. waiting for termination at the end of the task body. Should an exception be raised in the activation of one of the subtasks, then that task becomes completed. Should one of the tasks become completed during its *activation*, the exception *tasking_error* is raised upon conclusion of all of these tasks. A tasking supervisor should be prepared to raise the exception *tasking_error* in the master, before the first statement following the declarative part is executed.

3. It is possible to have a set of tasks created during the elaboration of a declarative part. Activation of such a set only takes place when the elaboration of the declarative part has been completed successfully. At the end of the activation of a set of tasks, the thread of control that executes the creation of the task should be made *executing* again. At the time of the activation of a set of tasks created by the same allocator or in the same declarative part, the supervisor should be notified for two purposes:

   - the task should be activated;
   - the thread of control, executing the master, should be suspended until all subtasks have been activated.

4. After the activation of the subtasks is successfully completed, the body of the task may start executing; its state is then *executing*.

5. Supervisor intervention is required to suspend the thread of control on reaching the end of a non-trivial master until all its dependent tasks have terminated. Precautions should be taken to allow the master to wait until all its dependent tasks have terminated, even after an exception was raised in the master. (The rationale for this is that an exception raised in the subtasks will not automatically propagate to the master.)

6. When a thread of control reaches the end of a task body, i.e. when it has finished the execution of the sequence of statements that appear after the reserved word *begin* in the task body, the task is defined to be *completed* (section 9.4 [LRM-83]); its state is

made *completed.* Reaching the end of a task body, whether all dependent tasks have terminated or not, indicates entrance to the *completed* state. All tasks with calls waiting for an entry of the task in the newly-completed state should be readied by raising the exception *tasking_error.*

7. *Termination* occurs if the task is in the completed state and it may terminate according to the rules of (section 9.4 [LRM-83]). To perform this termination, the supervisor should verify whether its master can proceed or not. This is important, since *termination* of a task can allow termination of the task executing its master.

Furthermore, a task can become *abnormal* by the execution of an abort statement in some other task (section 9.10 [LRM-83]). If a task has become *abnormal,* it should not participate in any rendezvous. A task which attempts to execute an exception handler and which has meanwhile become *abnormal,* must terminate rather than execute the handler. In the Ada— implementation, abnormality is handled by the tasking supervisor; no separate state *abnormal* hence is necessary.

6.2.3 Task communication

A second point that must dealt with is the modeling of a task communication implementation. First some general models for the implementation of rendezvous are discussed, then a state diagram and its transitions for modeling the rendezvous are discussed.

For the implementation of the Ada rendezvous three models are known from the literature:

1. the message-passing implementation;
2. the procedure-call implementation;
3. order-of-arrival implementation.

The Ada rationale [Rationale-79] proposes a so-called *message-passing* implementation. In this approach, the bodies of the *accept statements* are executed by the tasks owning the corresponding entries (the *server*† tasks). A call to an entry of a server task causes the parameters of the call and an identification of the calling task to be put into a mailbox associated with the entry. The message is queued and the sending process is suspended. The thread of control of the server task executes all accept bodies and performs all scheduling for entries. Whenever an *accept* or *select* is executed, the server queries the mailboxes of the open entries. If no message for any of the appropriate entries is available, the server suspends itself. When a message *is* available, however, the server executes an accept body for the entry. At the completion of the accept body the rendezvous is completed and the calling task reverts to the *executing* state.

Nassi et al [Nassi-80] have pointed out that this message-based approach could cause as many as three context switches in performing the rendezvous. First, the server reaches an accept statement, but since there is no pending call, it will be suspended. Another task,

† With a *server* task we mean a task owning an entry; the task serves requests from other tasks. The calling task, i.e. the task calling on the entry, is called the *consumer* task.

a consumer, then issues an entry call and is suspended while the server executes the rendezvous code. After having executed the rendezvous, the server finishes the rendezvous and the consumer is made *executing* again. If the static priority of the consumer is higher than that of the server, the server is switched out again.

Nassi et al [Nassi-80] discuss a solution to the problem of the many context switches for a single rendezvous. They suggest to allow the consumer thread of control to act as a *proxy* for the server by executing the latters accept body. The claim is that in most cases this approach results in at least one fewer context switch. This solution is generally known as the *procedure call* implementation.

Stevenson [Stevenson-80] discusses yet another implementation for the rendezvous model of the Ada language. This approach is called *order-of-arrival implementation.* It combines elements from both the procedure-call implementation and the message-passing-based implementation. In an order-of-arrival scheme it depends on the unpredictable execution sequence which thread of control must execute the accept body. If the server schedules a synchronizing statement and finds that a rendezvous is immediately possible, the server chooses to accept a call, to link the parameters and to execute the body. If no rendezvous is immediately possible, the server suspends itself. When a consumer issues an entry call and finds the entry being called not open or the server to be busy, the message-passing approach is taken and the parameters are queued as a message. Otherwise, the consumer proceeds to execute the accept body as in the procedure-call implementation.

Irrespective of how a rendezvous is implemented, support from the tasking supervisor is required for a number of operations. It is required not only for entering a rendezvous or leaving one, it is also required for propagating an exception that was raised during a rendezvous.

During task communication a task can be in one of the following states:

1. executing;
2. waiting for a caller;
3. waiting for a callee;
4. waiting for a caller or terminating;
5. participating in a rendezvous, either as caller (5a) or as callee (5b).

The possible transitions between the states are schematically given in figure 36. From the diagram, the following may be deduced:

1. Only a task *executing* can start or participate in any form of communication.
2. If, on reaching an accept statement, there are no pending calls, the task executing the accept statement must suspend itself (*waiting for a caller*). If the task which has encountered the accept statement has become abnormal, it must terminate (section 9.10 of [LRM-83]). Execution of a selective-wait contruct requires the supervisor to determine and respect the delay of the shortest duration (section 9.7 of [LRM-83]).

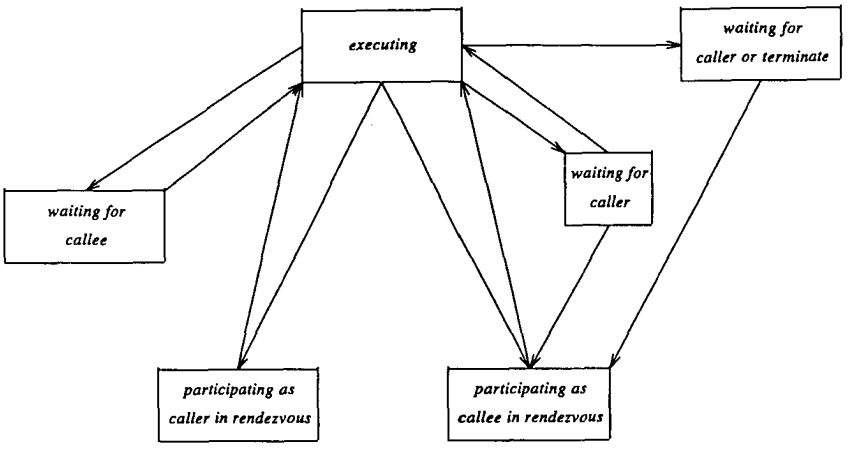


Figure 36. *Transitions of states in communication*

Execution of a selective-wait with an *else* alternative requires the intervention of the supervisor to check whether a rendezvous is immediately possible. If it is, it has a duty to execute the rendezvous there and then.

3. Section 9.5 of [LRM-83] requires that, on issuing an entry call, it should be inspected whether the owner of the entry is completed, i.e. in the state *completed* or *terminated*, in which case an exception *tasking_error* should be raised at the point of call in the calling task. A task issuing an entry call which is not immediately accepted is suspended (*waiting for callee*). The intervention of the supervisor is required for suspending the task as well as for queuing the request in the supervisor's data structures.

4. When a task has a selective-wait construct with an open terminate alternative (*waiting for caller or terminate*) and there are no pending calls, the task can terminate under the conditions given in section 9.4 of [LRM-83]. In that case a transition is made from this state to the state *terminated* in the other diagram (figure 35).

5. If the priorities of two tasks engaged in a rendezvous (*participating as caller in rendezvous* or *participating as callee in rendezvous*) the rendezvous is executed with the higher of the two priorities. A supervisor must ensure that the task that executes the body temporarily assumes the higher of the two priorities. The assistance of the supervisor is further required at the end of a rendezvous. An appropriate change in the state of the task must be made; moreover, the priority of the task executing the rendezvous should be restored and the partner in the rendezvous should be made *executing* again. If, within the accept body, an exception is raised which is not caught

by user-supplied exception handlers within that body, it must be propagated to both partners in the rendezvous.

6.3 A tasking supervisor for the Ada— compiler

A prototype tasking supervisor for the Ada— compiler was designed and implemented. A detailed description of the interface and the model of implementation has been given elsewhere [Katwijk-87b]. The tasking supervisor is highly target independent. Written almost completely in C, it provides support on UNIX systems as a single task. A version interfacing to a bare machine is in development. In this section the interface provided by the supervisor is discussed; implementation aspects are discussed in section 6.4.

According to the doublet model (chapter 5), a *task-type descriptor* is created on the elaboration of the task type. A task-type descriptor contains, among others, the static priority of the task, the starting address of the task-body code, some linkage information to allow proper task-object creation and a default storage size for run-time data of the task objects of the task type. The task-type descriptor contains data to allow the task creator to compute the static link required for addressing entities non-local to the task body. A task-type descriptor is implemented as a value of a private type:

type *task_type* **is private**;

Task-type descriptors are allocated and initialized under compiler control.

6.3.1 A supervisor interface for task management

Task objects are implemented as objects of an anonymous private type. They are identified by values of the private type *tdp* (task description pointer).

type *tdp* **is private**;

For the creation of a task object, the supervisor provides a procedure *create*.

procedure *create (m: master; c:* **in out** *tdp; tt: task_type; t:* **out** *tdp);*

Its parameters are:

- *m*, an identification of the tasks' master;
- *c*, a list header (each newly created task is linked in a list for the purpose of activation);
- *tt*, the descriptor of the task type of the task object;
- *t*, an identification of the newly created task, a value of type *tdp*.

Another parameter in the procedure *create* might have been specified as an indication for the storage size of the task object at run-time. In the current implementation we have discarded this parameter; instead a default storage size can be obtained from the task-type descriptor.

Entries within a task are identified by values of type *entry_desc*.

type *entry_desc* **is private**;

The type is given as a private type, the representation of values of the type is irrelevant to

the user. Baker [Baker-86a] gives an implementation suggestion; viz. as a private record type with a record component containing indications of the range in case of an entry family. Our technique has been to map the combination *entry* and *family element* into a single integral number.

In Bauner's interface [Bauner-80] a task object is described by a static descriptor taking the form of a (statically) initialized C structure. No explicit *create* function or procedure is provided. On the other hand, the supervisors described by Persch et al [Persch-84] and Falis [Falis-82] depend more strongly on the compiler. In their tasking implementation the compiler must take care of the allocation and initialization of task-object descriptors.

Our supervisor provides a procedure *activate* for the activation of a set of subtasks. A call to this procedure notifies the supervisor that a chain containing zero or more task objects must be activated.

```
procedure activate (c: tdp; s : boolean := false);
```

If required (according to section 9.3 of [LRM-83]), the procedure raises the exception *tasking_error*. The second parameter of the procedure is an indication of whether the required activation should take place at the end of a declarative part or during the execution of an allocator. This is merely an optimization; in several other proposals, e.g [Persch-84] and [Baker-86a], the same functionality was implemented with two separate procedures, one procedure to activate subtasks, the other to indicate to the supervisor that the elaboration of the declarative part (the activation) has been completed.

On run-time entry of a non-trivial master, a function *enter_master* must be called. The function returns a value of the private type *master*. This return value is used to identify the current master relative to the current thread of control. A descriptor for a master construct is implemented as a structured value which is kept on the activation-record stack. A value of type *master*, implemented as a pointer, is used to refer to such a master-description structure.

```
type master is private;
null_master : constant master;  -- null element;

function enter_master (old_master: master) return master;
```

The function takes as its parameter the unique reference to the old master description.

On exit from a non-trivial master, the function *exit_master* is called. On a call to *exit_master* the tasking supervisor suspends the current thread of control until all dependent tasks shall have been terminated. The specification of the function has the form:

```
function exit_master return master;
```

The function returns the identification of the master construct that was the current master in the prevailing construct. For each task descriptor, the supervisor maintains a *current master* indication. Therefore, the supervisor is able to let a thread of control leave a master without explicit indication which master will be left.

Each master construct is extended by a compiler-generated exception handler to allow the supervisor to suspend the thread of control at the end of a master, even when an exception is raised with the master. As an example, consider the block

```
declare
 ...
begin
  .....
end;
```

The translated form of this code looks schematically

```
declare
  ...
begin
  ...
exception
  when others => exit_master; raise;
end;
```

A task body is mapped onto a procedure, although it is never called as such (it is automatically activated by the tasking supervisor). To catch exceptions propagated from within the body, the generated code contains a compiler-generated exception handler.

As soon as the task body becomes *completed*, the supervisor procedure *terminate* is called to take care of the actions required for task termination. Now consider an example of a task body and an outline of its translation.

```
task body A is
-- declarations
begin
-- statements
end;
```

The translated form of a task body is the translated form of a procedure *task_body*.

```
procedure task_body is
  chain: tdp;          -- the chain of subtasks
  m : master;          -- maintaining masters
  dummy: master ;
begin
  m := enter_master (null_master);
-- now for the declarations
-- and the statements of the task body
  declare
    ...    -- local declarations
    ...    -- elaborate them, chain task objects
  begin
```

```
    -- notify supervisor of arrival here,
    -- activate subtasks, propagate tasking_error if required
    activate (chain, true);
    sequence of statements
  exception
    when ...=>    -- user supplied handler
  end;

  dummy := exit_master;              -- dummy assignment
  terminate;      -- try to terminate
exception
  when others =>            -- compiler generated handler
       dummy := exit_master;
       terminate;
end;
```

Prior to entering a rendezvous or entering an exception handler, a task should make sure that is has not become *abnormal.* According to section 9.10 of [LRM-83], a task that has become *abnormal* should terminate in any case. Checking for abnormality can be done by the tasking supervisor: as soon as a task becomes *abnormal,* the supervisor decides to terminate the life of a task. The handling of *abnormal* tasks is different in other proposals: Bauner et al do not consider *abnormal* tasks at all; Falis provides a function *check_abnormal,* although the option is kept open to have the supervisor deal with *abnormal* tasks and Bayan et al also propose a tasking supervisor procedure to execute the abort.

A supervisor interface for handling rendezvous

Rosen [Rosen-83] describes a small supervisor interface for the support of task communication. The interface consists of only three procedures. The interface for handling rendezvous in the Ada— compiler is based on Rosen's proposal. All forms of entry call are mapped onto a single form, a call to a single tasking supervisor procedure. Similarly, all forms of selective-wait statements, including a stand-alone accept statement are mapped onto calls of another single tasking supervisor procedure. The form of all entry calls is unified to a *timed entry call.* An ordinary entry call

```
E
```

is mapped onto a timed entry call

```
select
  E;
or
    delay INFINITE;
    null;
end select;
```

(*INFINITE* is a particular value, to be interpreted as an infinite delay.) Similarly, a

conditional entry call

```
select
  E;
else
  sequence of statements
end select;
```

is mapped onto a timed entry call:

```
select
  E;
or
  delay 0;
  sequence of statements
end select;
```

The standard form in which a timed entry call appears is:

```
select
  E;
or delay T;
  statement sequence
end select;
```

An entry call is implemented by a call to the supervisor procedure *call_rdv*.

```
function call_rdv (t : tdp; e: entry_desc; d: duration; p : system. address) return boolean;
```

In this function:

1. *t* is a task pointer, identifying the task owning the entry to be called;
2. *e* is the identification of the entry;
3. *d* indicates the delay. According to [LRM-83], the delay should be able to take values up to at least 86,400 units of time;
4. *p* is the address of a block of data where the actual parameters can be found. Given *p* as base address, the callee can address individual parameters.

The function returns a boolean value. This value indicates whether the rendezvous has taken place or not.

Translated in terms of a call to a supervisor function, the translation of a timed entry call looks like:

```
if not call_rdv (t, e, d, p)
then
  statement sequence
end if;
```

The use of a function is disputable since the function clearly has side effects. This use

must be considered as a concession to the C implementation; in C, it is easier to deal with a function result than with an *out* parameter.

Selective-wait constructs are always mapped onto a call to a single supervisor procedure. Even a single, stand-alone, accept statement is mapped onto a selective-wait construct as demonstrated below.

```
accept E do
  statement sequence
end;
```

is elaborated as though it was:

```
select
  accept E do
    statement sequence
  end;
or
  delay INFINITE;
  null;
end select;
```

The description of the run-time modeling of a selective-wait construct is best given in terms of Ada type declarations. Enumeration literals are assumed to have been defined to indicate the characteristic of the alternative to be described.

```
type kind is (closed, delay_alt, accept_alt, terminate_alt);
```

Each alternative is described by a variant of a variant record; the definition of the whole record speaks for itself.

```
type alternative_desc (k: kind) is record
  case k is
    when closed | terminate_alt
            => null;
    when delay_alt
            => del: duration;
     when accept_alt
            => entry: entry_descr;
  end case;
end record;
```

to go with the following type definition:

```
type entry_vector is array (integer range <>) of alternative_desc;
```

The supervisor procedure implementing a selective-wait construct is *start_rdv*:

```
procedure start_rdv (n:  integer;
                     d: in entry_vector;
                     t:  out tdp;
                     c:  out integer;
                     p:  out system. address);
```

The parameters of this procedure are:

- *n*, an integer indicating the number of alternatives (either open or closed) of the selective-wait construct;
- *d*, a vector of alternative descriptors. For each alternative of a selective-wait construct the vector contains a description;
- *t*, an *out* parameter yielding the identification of the partner in the rendezvous†;
- *c*, an *out* parameter yielding the order number of the selected alternative;
- *p*, an *out* parameter yielding the address of the caller's actual parameters.

The *out* parameters *t* and *p* only have defined values in case of a successful rendezvous.

The execution of a rendezvous is terminated by a call to to *end_rdv* procedure.

```
procedure end_rdv (x: system. address := 0);
```

A call to this procedure informs the supervisor that the rendezvous has been completed and that both tasks involved in the rendezvous must be made *executing* again.

The procedure is called with a non-zero parameter when an exception must be passed to both tasks involved in the rendezvous. The parameter value then identifies the exception (see section 5.4). The translated code for an *accept* body looks schematically:

```
begin
  S1; -- .... statements from the body
  end_rdv;
exception
  when others => end_rdv (current_exc);
end;
```

The compiler allocates and initializes an array object of type *entry_vector* in the context of the caller. This object contains a description for each alternative description; this object is passed as a parameter to the procedure.

Consider the example of the following selective-wait construct

---

† In fact this should be seen as an extension; there is no need whatsoever from the point of view of the language semantics for the server task to have any knowledge of the identity of the calling task.

```
select
  when C1 =>
   accept rdv1 do
     S1;
   end;
or when C2 =>
  accept rdv2 do
       B. rdv3;  -- entry call to entry
                 -- rdv3 in some task B
  end;
or when C3 =>
  delay T1;
    S3;
or when C4 =>
  delay T2;
    S4;
end select;
```

The code generated by the Ada— compiler contains explicit initializations of the alternative descriptors:

```
alternative_descriptor : entry_vector (1 .. 4);
```

The initialization of the descriptors is, omitting irrelevant details:

```
if C1
  then alternative_descriptor (1) := (accept_alt, entry_1);
  else alternative_descriptor (1) := (k => closed);
end if;
if C2
  then alternative_descriptor (2) := (accept_alt, entry_2);
  else alternative_descriptor (2) := (k => closed);
end if;
if C3
  then alternative_descriptor (3) := (delay_alt, T1);
  then alternative_descriptor (3) := (k => closed);
end if;
if C4
  then alternative_descriptor (4) := (delay_alt, T2);
  else alternative_descriptor (4) := (k => closed);
end if;
```

The initialization code can be shortened (at least when written in the Ada language) by having the variable *alternative_descriptor* initialized. The code in the C based implementation is in fact shorter. C allows procedures to have a variable number of parameters and the elements of the alternative descriptors themselves are passed as

parameters to the *start_rdv* procedure.

The code for the implementation of the selective-wait construct is:

```
start_rdv (4, alternative_descriptor, t1, c, p);

case c is
  when 1 =>
  begin
    S1; end_rdv;
  exception
    when others => end_rdv (current_exc);
  end;
  when 2 =>
  begin
    dumm := call_rdv (B, rdv3, INFINITE, 0);
    end_rdv;
  exception
    when others => end_rdv (current_exc);
  end;
  when 3 => S3;
  when 4 => S4;
end case;
```

For a supervisor encoded in the Ada language, the first parameter is superfluous; its value can always be computed as an attribute from the second parameter. That it appears at all is a concession to the C implementation.

It does not seem sensible to distinguish between various forms of selective-wait constructs and to provide different procedures for each of the classes distinguished as though, e.g. Bauner et al. do. If, in the example given above, the conditions C3 and C4 both yield the value *false*, the construct turns out to be a simple selective-wait construct with only *accept* alternatives.

The overall structure of our interface resembles that proposed by Rosen. A divergence of our interface is the way alternatives for a selective-wait construct are described. In our supervisor we describe the *alternatives*, similarly to the approaches by Baker et al in the FSU compiler and by Persch et al in the Karlsruhe compiler. Rosen [Rosen-83] suggests to describe the *entries* to which the accept alternatives belong. A (possible) delay is passed as an additional parameter to the *start_rdv* procedure. When dealing with an open terminate alternative a flag is set. This approach has two consequences:

- as a minor consequence, a selective-wait construct with two or more open delay alternatives requires additional compiler-generated code to determine the smallest delay and its associated alternatives;

- as a major consequence, it is not possible to describe a selective-wait construct with two or more (open) *accept* alternatives for the same entry as is required by the language (section 9.7 of [LRM-83]).

Baker et al [Baker-86a] suggest a description of the open *accept* alternatives as a parameter. One more parameter is required in order to pass the smallest delay value to the supervisor. A drawback of this approach is that the compiler must generate code to dispatch the smallest delay value whenever more than a single delay alternative occurs. The gain in having to pass a shorter vector containing only descriptions of open alternatives does not seem worthwhile.

In the Ada— supervisor the handling of *nested* accept statements is transparent to the user of the interface. No special precautions must be taken by the user in case of a nested rendezvous.

In particular, on tasking communication, the interface proposed by Falis is different from other proposals. In Falis' scheme the transformed program must indicate which task must execute the accept body. The interface suffers from having different procedures for starting a rendezvous by the consumer and by the server. Additional functions are required to check whether a rendezvous may take place immediately or not.

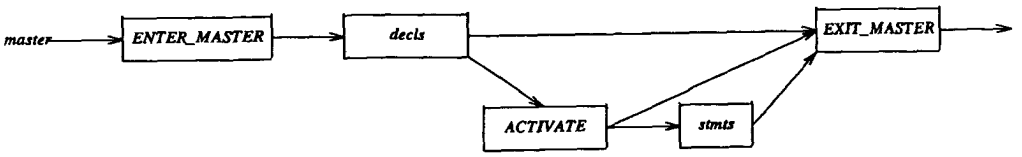
6.4 A schematic overview

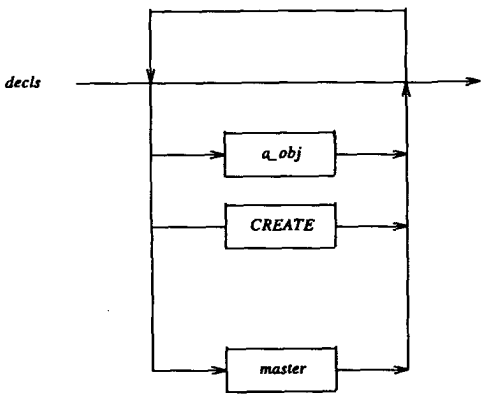
Another way to look at the tasking supervisor is to specify the legal sequences of supervisor calls. It will be intuitively clear that not every sequence of supervisor calls is correct: a call to *exit_master* is only legal after a call to *enter_master* has taken place. In this section we describe the legal ordering of the various supervisor calls. Since the sequences of calls can be interpreted as sentences in a formal language, a context-free grammar is given as a description. In a sense, the description in this section is complementary to the description in the previous sections of this chapter.

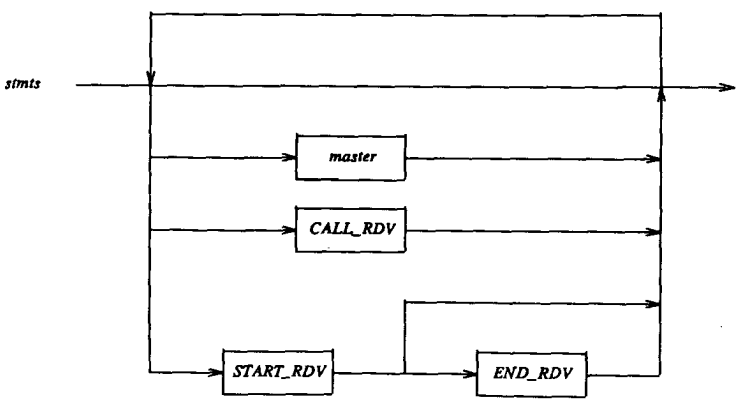
The description is couched in terms of syntax charts; the following terminology is used:

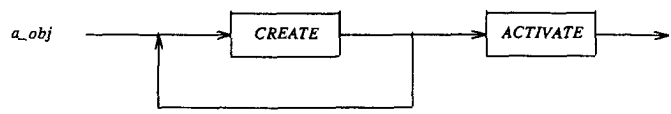
1. A notion in (italic) upper case, say *X*, is used to indicate a call to the supervisor procedure *X*. An upper case notion is terminal in our syntax; it only appears in applied positions.
2. A notion in (italic) lower case, say *x*, is used as non-terminal. In a defining position it is the label of a syntax chart; in an applied position it indicates the applied occurrence of a chart.

The charts are given below. The main chart is labelled *tb*, describing a task body.


master
ENTER_MASTER
decls
EXIT_MASTER
ACTIVATE
stmts


decls
a_obj
CREATE
master


stmts
master
CALL_RDV
START_RDV
END_RDV



tb The tasking supervisor calls in a task body are calls made in a master construct followed by a call to the procedure *terminate.*

master In a master construct a call is made to the procedure *enter_master.* Next, it is possible that supervisor calls are made during the elaboration of the declarative part (*decls*). If an exception is raised during the elaboration of such a declarative part, control is transferred to the compiler-inserted exception handler. This exception handler contains a call to the procedure *terminate.* Otherwise, the normal flow of control will lead to a call to the procedure *activate* and, eventually, to a call to *exit_master.*

decls During the elaboration of declarations, supervisor procedures can be called in several different ways:

- the procedure *create* is called on for the elaboration of ordinary task objects; the task object will be chained for later activation;
- expressions containing allocators may be evaluated on the elaboration of the declarative part; activation is then under control of the allocator;
- during the elaboration of a declarative part, statements may be executed.

stmts During the execution of statements, supervisor calls can be made:

- when executing a non-trivial master;
- when performing any entry call;
- when in a server, by performing a call to the procedure *start_rdv.* If a rendezvous turns out to be effectively made, the procedure *end_rdv* mus be called, otherwise, it may not.

a_obj The evaluation of an allocator may lead to the creation of task objects. The activation of these task objects takes place at the end of the operation of the allocator. Any call to the procedure *create* must therefore be followed by a call to the procedure *activate.*

6.5 The implementation of the Ada— tasking supervisor

In this section the general structure of the implementation is discussed; implementation details are omitted, however. A more thorough description of the implementation, including a detailed description of the C implementation, is given in [Katwijk-87b].

The implementation of our tasking supervisor uses events and time-slicing. Its design was strongly influenced by the supervisor proposals of Rosen [Rosen-83] and by Baker and Riccardi [Baker-84], [Riccardi-85], [Baker-86a].

Limited amount of space preclude a full description of our implementation. Natural language therefore takes the place of an Ada formulation in what follows.

6.5.1 Task creation and task-description blocks

Each task object is described by a *task-description block.* A task-description block contains the data required for the execution of the code of the task. As such it resembles task-description blocks in other proposals; it even resembles the UNIX *user* and *proc* structures. A complete definition and description of a task control block is not within the scope of this chapter; for details the reader is referred to [Katwijk-87b].

A task-description block contains i.a.:

- machine-dependent data; e.g. the values of the registers to be initialized for the execution of the task object;
- pointers to the various chains and lists a task object may be linked to;
- status information of the task;
- administrative data for the management of its various non-trivial masters.

Furthermore, the task-description block contains a region of space for the activation records at the execution of the task, formalizable in Ada as:

```
type task_description_block (size_of_stack:  positive)
is record
   ...
end record;

type tdp is access task_description_block;
```

where the discriminant constraint indicates the size of the run-time stack segment at the execution of the task.

Tasks are created by a call to the supervisor procedure *create.* For each newly created task a task-description block is made. Creation of a task entails, i.a. allocation of a work space and initialization of the task-description block. A specific target-dependent issue is the set-up of the activation-record stack for a task on its creation. A task body is translated as if it were a procedure without, however, being explicitly called. When the task object is created, an environment within the task object is built such that the result is indistinguishable from those that would have resulted had the task executed a scheduler call. Therefore, at task creation, the activation-record stack of the task object must be filled with data simulating such an environment. Setting-up such an environment is rather target-dependent.

An Ada model of the *create* procedure might read:

```
procedure create (m: master; c: in out tdp; tt: task_type; t: out tdp) is
begin
  t := new task_description_block (...);

  enter_list (t, m);  -- enter t in master m
  c := link_list (t, c); -- enter t in activation chain
end;
```

The functions *enter_list* and *link_list* are self-explanatory. The current task, i.e. the task that is governing the current thread of control, is always identified by a global variable:

```
myself: tdp;
```

6.5.2 Low-level facilities

In the tasking supervisor, low-level scheduling is implemented by a form of signal/wait mechanism. The low-level primitives are a *signal* procedure and a *wait* procedure operating in tandem. The set of signals in the tasking supervisor is fixed, they are - in terms of a notational enumeration type -:

```
type signal is (timer_event,
                .....,
                to_be_engaged,      -- waiting for a rendezvous
                kill_event);        -- terminate processing event
```

The procedure *signal* sends a signal to the task object that is identified by the first parameter. The second parameter of the call is an encoding of the signal being sent. The specification of the procedure *signal* reads:

```
procedure signal (T : tdp; s : signal);
```

The thread of control issuing the signal continues unchanged, i.e., no re-scheduling takes place.

The procedure *wait* has been provided with a *mask* as one of its parameters. This mask, implemented as a boolean vector represents an encoding of the signals that will be reacted upon. The mask is expressed as a value of type *signal_mask*

```
type signal_mask is array (signal) of boolean;
```

Some standard mask values that are current in our implementation are defined below:

```
anything: constant signal_mask := (others => true);
-- react on any signal
nothing: constant signal_mask := (others => false);
-- react on no signal at all
kill: constant signal_mask := (kill_signal => true, others => false);
-- react only on a kill signal
engaged: constant signal_mask := (to_be_engaged => true, others => false);
-- react on a signal to_be_engaged
timer: constant signal_mask := (timer_event => true, others => false);
-- only react on a timer signal
```

The procedure *wait* suspends the current thread of control, until either:

- a signal, for which the corresponding mask value is *true*, is received,
- or, the delay expires

whichever occurs first.

The specification of the procedure *wait* reads:

```
procedure wait (t: delay; sm: signal_mask);
```

Specific messages to the procedure *wait* are encoded by particular delay values; *0* and *INFINITE* are examples. If a re-scheduling activity is required for whatever reason, a delay value *0* is chosen. If a task insists on waiting for a signal, regardless of the delay it may cause, a delay value *INFINITE* is chosen. A task that issues

```
wait (INFINITE, nothing);
```

is never rescheduled. Each task-description block contains a component *last_event* in which an encoding of the signal most recently received is maintained.

6.5.3 Masters, entry and exit

The task objects depending on a master are chained together. The master description is a record-like structure:

```
type master;

type master_description_chain is access master;

type master is record
   child:    tdp;
   number:   integer;
   next:     master_description_chain;
end record;
```

Master descriptions are kept on the activation-record stack of the enclosing task body. Each master description contains three components:

- a *child* component contains the header of a chain in which the *children* of the task (i.e. its depending tasks) are chained;
- A *number* component indicates the number of the tasks in the *child* chain;
- A *next* component contains a link to the master description of the enclosing master, if any.

Each *task-object description* (subsection 6.5.1) contains a component *current_master* that contains an access value to the description of its current master.

The implementation of *enter_master* (subsection 6.3.1) is straightforward, a new master description element is allocated†:

```
function enter_master (om: master) return master is
begin
   myself. current_master :=
        new master_description' (null, 0, om);
   return myself. current_master;
end;
```

The implementation of *exit_master* (section 6.3.1) is more complicated: when leaving a non-trivial master, the thread of control executing that master must be suspended until all dependent tasks have terminated. The implementation of the function *exit_master* is schematically:

```
function exit_master return master is
begin
  for i in myself. current_master. number
  loop
    wait (INFINITE, kill);
  end loop;

  for t in all_dependent_tasks
  loop
    signal (t, kill_signal);
  end loop;

  myself. current_master := myself. current_master. next;
  return myself. current_master;
end;
```

The thread of control executing the task is suspended in the first loop until a signal is received from all dependent tasks indicating their ability to terminate. The second loop causes a *signal* to be sent to all dependent tasks. They can take appropriate *terminate* actions after receiving the signal.

---

† In our implementation the master descriptions are kept on the activation record stack; the extent of the descriptions is known.

6.5.4 Task completion and termination

The procedure *terminate* takes care of all actions required for termination after completion. The compiler ensures that a call to this procedure is always preceded by a call to the procedure *exit_master*. Thus, the task for which the thread of control executes the procedure *terminate* is guaranteed to have no living children. The procedure *terminate* operates in several steps: it signals the master of the task that the task in question task is able to terminate (by completion) and it waits for an acknowledgement. Finally, it releases the resources and suspends itself forever.

The parent task will sent an acknowledgement as soon as the conditions under which the task actually can terminate are fulfilled. A schematic version of the implementation of the procedure *terminate* is given below:

```
procedure terminate is
begin
  signal (myself. parent, kill_signal);
  -- kill here all tasks that are waiting for
  -- a rendezvous with this one
  -- wait for an acknowledgement from
  -- parent
  wait (INFINITE, kill);
  -- do all processing required to clean up the mess
  -- and to return resources
  -- allocated for the task
  wait (INFINITE, nothing);  -- no return
end;
```

6.5.5 Rendezvous

Several functions and procedures are involved in the implementation of a rendezvous. Our discussion will restrict itself to the two most important procedures, *call_rdv* and *start_rdv*.

A task entry is identified by a pair made up of a pointer to the callee's task-description block and an entry descriptor, represented by an integral value.

```
type entry_desc is range 1 .. 127;
```

An outline of the implementation of the function *call_rdv* is given below:

```
function call_rdv (t : tdp;  -- task to be called
          en:  entry_desc;   -- entry
          d:  delay;
          p:  system. address) return boolean
is
begin
  if not t' callable
```

```
    then
      raise tasking_error;
      -- abortive return
    end if;

    -- chain myself, the caller, in the queue for the
    -- entry
    rdv_chain (myself, t, en); -- the variable myself is global to the call
    signal (t, to_be_engaged);

    wait (d, engaged or timer);
    -- the callee will send a to_be_engaged signal
    -- when it has honoured the call
    -- otherwise the timer causes a timeout signal to be sent
    if myself. last_event = timer_event
    then
        -- time out, no rendezvous
        return false;
    else
        return true;
    end if;
  end;
```

In the above, a check first is made that the called task is indeed *callable.* Next, the current task (identified through *myself*) is chained into the rendezvous queue of the called task (*rdv_chain*) and a *signal* is sent to the callee. If the latter is waiting for a rendezvous, it is woken up and rendezvous processing may continue.

The procedure *wait* is called in order to relinguish the (current) processor. This procedure causes the thread of control executing the call to this procedure to be suspended until either a time-out has occurred or a signal is received that indicates that a rendezvous has taken place. Action is then taken, depending on the signal received.

As a procedure *start_rdv* is by far the most complex within the tasking supervisor. Prior to calling the procedure, a vector must be filled with complete data for the description of the alternatives of the select construct. This vector is passed as a parameter to the procedure.

The implementation of *start_rdv* falls into three parts. In the first part, the structure of the selective-wait construct is chosen and justified. From the input supplied, the smallest delay value is computed and it is checked that at least one alternative is open. If not, an abortive exception is raised and propagated. In the second part it is inspected for which open alternative a caller is waiting, if any. If such a caller is found (there may be more than one), the *out* parameter *c* is set to the sequence number of the alternative selected and a successful return from the procedure is made. (This value is used to select the alternative to be executed.) Finally, the third part deals with the selection of a *terminate*

alternative or the selection of a *delay alternative*, in either case no return takes place.

A schematical overview of the implementation of *start_rdv* is given below:

```
procedure start_rdv (n:   integer;     -- number of alternatives
                     d:   entry_vector;    -- alternatives descriptor
                     t:   out tdp;         -- partner task
                     c:   out integer;     -- number of selected alternative
                     p:   out system. address) is

  no_one: boolean := true;
  delay_time: duration := INFINITE;
  term_alt : boolean := false;
  begin
    for i in 1 .. n
    loop
      declare
        X : alternative_desc renames d (i);
      begin
        if X. kind = delay_alternative
        then
          if X. del < delay_time
          then
            delay_time := X. del;
            sel := i;
          end if;
        elsif X. kind = terminate_alt
        then
          term_alt := true;
          no_one := false;
        elsif X. kind = accept_alt
        then
          no_one := false;
        end;
    end loop;
    if no_one
    then
      raise tasking_error;  -- no open alternative
    end if;
  -- In the second loop, the real action takes place
    loop
    -- First look for an open accept alternative and see whether
    -- someone is waiting
      for i in 1 .. n
      loop
```

```
        declare
          X : alternative_desc renames myself. alt_desc (i);
        begin
          if X. kind = accept_alt and then
              not is_empty_queue (myself, X. entry)
          then
          -- return the number of the selected alternative
            c := i;
            return;
          end if;
        end;
  -- If we are here, no open accept alternative is serving
  -- a request
  -- If the kind of the selective-wait construct is not
  -- a terminate type, then wait until someone signals us:
    if not term_alt
    then
      wait (delay, anything);
      -- any signal wakes us up. Assume for simplicity that
      -- either a timer_event event or a to_be_engaged event will wake us:
      if myself. last_event = timer_event
      then
        c := SEL;
        return;
      end if;
      -- If we are woken up by a timer_event event, i.e. a time-out,
      -- then return the number of the alternative with the smallest
      -- delay.
      -- If we are woken up by a to_be_engaged event the dispatch loop will
      -- be executed again, waiting for a call to an open accept entry;
      -- in case of a terminate-type construct, keep it
      -- simple and let a routine (here unspecified) do the job.
    elsif term_alt
    then
      check_terminate (myself. my_father);
      wait (INFINITE, anything);
      if myself. last_event = to_be_engaged
      then
        reactivate (myself. myfather);
      elsif myself. last_event = kill_event
      then
        perform terminate actions and die piecefully
        wait (infinite, nothing);  -- no wake up
      end if;
    end if
```

```
  end loop;
end;
```

7. The Ada— program library

The Ada language supports a safe separate compilation facility. The implementation of this facility requires a *program library* in which compilation units are kept. The Ada language definition ([LRM-83]) does not specify how a program library should be implemented, although some functional properties of program libraries are prescribed.

Program libraries are the centre of interest of many people. The questions to be answered go from simple questions, such as how to keep the library up to date at a reasonable cost, up to questions of a more pronounced software-engineering nature, such as how versions and mile-stone configurations are to be maintained. One of the reasons that research in the area of program libraries and extensions will continue is the lack of full-grown APSEs.

In this chapter the design and implementation of the program library for the Ada— compiler is discussed. First the requirements, imposed on a library by the Ada language are discussed; then a survey of the existing literature is given. The survey is followed by an overview of program-library models and the various implementations for the Ada— compiler.

7.1 Requirements for an Ada program library

Chapter 10 of [LRM-83] gives a description of the functional requirements to be imposed on a separate compilation facility for Ada. In each compilation, one or more compilation units may be presented to the compiler. A compilation unit consists of an optional context clause, followed by either a *library unit* or a *secondary unit.* A library unit is a package specification or a subprogram specification.

A secondary unit is either the (separately compiled) proper body of a library unit (the *library-unit body*) or the subunit of another compilation unit. A subunit is used for the separate compilation of the proper body of a program unit declared within another compilation unit.

The combined library units constitute (part of) the program library.

The context of each compilation unit during compilation as well as during elaboration is defined by the predefined package *standard*, any other compilation unit listed in the context specification or the specification of the parent's unit name, and recursively the contexts they invoke.

Several issues are stated clearly in chapter 10 of [LRM-83]:

- if any error is detected while attempting to compile a compilation unit, then the attempted compilation is rejected. The compilation thus has no effect on the program library (any recompilation counts as a compilation);
- a compilation unit can only be compiled after all library units named by its context clause have been compiled. A secondary unit that is a library-unit body must be compiled after the corresponding library unit. Any subunit of a parent compilation unit must be compiled after its parent.

The second rule implies a partial ordering for the compilation of compilation units.

On the compilation of a unit *X*, the units that must be consulted by the compiler for the current compilation are determined as follows:

- if the compilation unit is a library unit, then each name mentioned in a *with clause* must be consulted. This process is closed under transition implying that each library unit mentioned in a *with clause* of the library unit being consulted must be consulted as well;
- if the compilation is a library-unit body then the corresponding specification and the declarations from that specification are to be incorporated as well as all declarations from units mentioned in a *with clause* (also closed under transition);
- if the compilation unit is a subunit, then the relevant part of the parent compilation unit is to be consulted. This process being closed under transition, the relevant data of the parent units must be consulted as well.

7.2 A survey of the literature

A number of publications describes various aspects of program libraries. Early descriptions have been given by Dausmann et al [Dausmann-80a], van Someren [Someren-82b] and Rasmussen [Rasmussen-82]. These descriptions deal with separate compilation and an efficient computation of dependency relations. Later publications, e.g. [Narfelt-85], [Schefstrom-86], and [Carney-86], focus on features and facilities of the program library, so as to promote its use in software engineering.

Dausmann et al were the first to give a description of an elementary program library. Their description is in three parts. The basic structure of the program library of the Karlsruhe Ada compiler is discussed in [Dausmann-80a]. Efficient recompilation checks are discussed in [Dausmann-80b] and the problem of re-using library units in other libraries is discussed in [Dausmann-80c].

Van Someren [Someren-82b] describes a first proposal for the program library of the DAS compiler. The approach proposed was the basis for the first implementation of a program library (see the discussion in section 7.4).

Briggs [Briggs-84] describes two different implementations of a program library of the York compiler. The first library was a closed one, similar to the one described in subsection 7.4.1. The second implementation relies on facilities of the underlying UNIX system. The approach in the first implementation model is to implement the library as a set of files with a single catalogue file to describe them. For each representation of a compilation unit, a file is used; furthermore, the catalogue file contains descriptions of all representations and the files holding them. The first objection to this approach was that the use of the library followed a completely different paradigm to the use of other tools in the UNIX system. Therefore, it proved difficult to integrate the tools with other tools in the UNIX environment.

The second objection was that sharing of library units among libraries was impossible. Furthermore, it was impossible to have several versions of a body for the same

specification. According to the author, [Briggs-84], these drawbacks led to systems that were very difficult to maintain.

The second program library described by Briggs was more in line for use with other UNIX tools. The library manager heavily relies on the UNIX *make* utility [Feldman-79b]. Compilation is almost completely under control of the program 'make'. Thus make-files can be generated by a special tool contained in the library; they can therefore be modified at will by the user. Although this approach is rather flexible, its major drawback is that the consistency of the library is critically dependent on the collective whim of all users.

Lettvin [Lettvin-82] describes the program library supported by the BreadBoard compiler. The library is built as a set of UNIX files with a separate *dictionary* file. The Ada linker and other Ada-specific tools can access the units by consulting the unique associated dictionary.

Similarly to other table-stored data on UNIX systems (e.g. the password file) the dictionary is a simple ASCII file, each line describing a compilation unit. Consider the example of the following lines taken from such a system:

1000, BAAAAA
IO_COMMON:PH:I3:NA:394291731:0::::
TEXT_IO:PH:T2:NA:34291677:0::::

The second line in this example states that a library unit *IO_COMMON* is part of the library. *PH* indicates that the unit is a package specification. The compilation date is stored as a single number, *394291731*. The library unit is identified by *I3*. Similarly, the third line in this example states that a library unit *TEXT_IO* is part of the library.

Based on data in the dictionary file, the Ada compiler and the Ada linker are capable of determining the names of the files that are required as argument for a compilation or for the construction of an executable piece of code. From Lettvin's description, consistency of the library would not appear to be guaranteeable.

Although no detailed description is given of the library of the Verdix compiler [Verdix-85], it appears that the structure of the program library for this compiler is similar to the structure of the program library for the BreadBoard compiler. The library consists of separate files for all representations of compilation units and an - ASCII-encoded - catalogue file. The structure of the catalogue file resembles the structure of the catalogue file of the BreadBoard compiler.

Narfelt et al [Narfelt-85] discuss an approach in which the scope of the program library is extended. According to the authors the ultimate goal of their work is to evolve from a simple program library towards a full-grown database for a software engineering environment. Their approach is based on a multi-dimensional extension of ordinary plane graphs to control the relations between the various compilation units. The authors' extensions comprise: the *typing* of edges and a layering of the graph. Each layer of the graph can be used to represent different versions of (parts of) the program library. For a single, isolated, layer the term *sub-library* is used. The extensions of the program library that are discussed by Narfelt et al have been implemented in the library manager of the

Telesoft Ada Compiler.

Schefstrom [Schefstrom-86] describes a logical continuation of the work of Narfelt et al. The program library was extended by configuration-management tools, after which the need for a version-management system arose. Schefstrom proposed the use of RCS [Tichy-85] together with the program library of the Telesoft Ada compiler.

The AIE compiler project included a full AIE library; it was due to be completed in November, 1985. The design of the library and some experience of its use are discussed by Carney [Carney-86]. This library differs from other systems (in particular from the ALS system [Thall-82]) in that it is primarily concerned with compilation units rather than with source-object pairs. The source is fully reconstructable from the unique intermediate DIANA representation stored in the program library. Therefore, not maintaining explicit source modules in the data base hardly constitues a restriction. The AIE library consists of a set of files and an associated catalogue. The catalogue identifies its constituent units in every significant detail. A *collection* is a storing place for the compiled units and the catalogues identifying them. A catalogue is a user's one and only view of the library. Catalogues can be frozen; once frozen their contents can be shared and used by others. A frozen catalogue makes resources available to other catalogues. For this reason, once frozen it is called a *resource catalogue.* All *resource catalogues* are potentially available to all users through transitively closed catalogue links. The user operates on a primary catalogue; the constellation of a primary catalogue together with its linked resource catalogues form a single conceptual program library, the linking being invisible to the user.

The problem of maintaining a program library within a full-grown APSE is outside the scope of this thesis. For completeness sake, the library structure of the ALS as described by Thall [Thall-82] and some library issues in the Eclipse environment [Pierce-85] are mentioned.

The design of the ALS was based upon two major principles. The first one was the retaining of the principles of the UNIX operating system. UNIX was chosen because its file system has a relatively simple structure, it was well received by its user community and it had a good record of being rehostable. The second major guiding principle was derived from the United Kingdom study on Ada-support systems. The structure of the ALS data base is akin to the structure of the UNIX file system; it too is tree-structured. It's database distinguishes two additional node kinds, viz. *revision sets* and *variation sets.* When an attempt is made to modify a frozen version, a new *revision* of the file is created. Revisions are automatically numbered. In order to support this the ALS introduces the notion of *file revisions.* Revisions may be introduced for many different reasons, amongst them modifications to adapt to different hosts and different target computers. For help in the identification and selection of revisions, ALS provides the notion of a *variation set.*

Pierce [Pierce-85] discusses the Eclipse, a self-styled *Integrated Project Support Environment.* The scope of an integrated project support environment exceeds that of the scope of an APSE. One of the differences is that an IPSE is not necessarily mono-lingual as an APSE is by definition. A central facility of the Eclipse environment is the database.

A program library is itself a database object. Furthermore, each compilation unit produced by a compilation is also represented as a database object. A number of versions of the program library itself exists. These versions result from successive compilation units entered into it. In principle, each compilation generates a new library, although there are features preventing the number of program libraries to become excessive.

7.3 The compilation and the elaboration order of compilation units

The compilation order of compilation units

Many algorithms have been published which were meant for the establishment of a compilation order. These algorithms have a single base of an algorithm determining the transitive closure of a binary relation.

We define the binary relation *directly dependson* on a pair of compilation units *A* and *B* such that *A directly dependson B* as:

- *B* is named in a *with clause* of *A*, or
- *A* is a body and *B* is its specification, or
- *B* is a body and *A* is a *subunit* of that body.

The relation *dependson* is the transitive closure of the relation *directly dependson.*

A binary relation such as *dependson* defines a partial ordering over the compilation units in a program library. Given any two compilation units *A* and *B*, *B* should be compiled before *A* is compiled *iff A dependson B* holds. Given the relation, it is easy to generate either a *makefile* or to compile directly in the correct order any set of compilation units. An outline of an algorithm for the recompilation of a unit *x* and all units on which *x* is dependent is given below:

```
compile (x)
begin
  forall units y mentioned in with clauses of x
  loop
    if is_invalid (y)
    then
      compile (y);
    end if;
  end loop;

  apply the compiler to unit x

  if compilation succeeded
  then
      mark x valid;
  end if;

  forall compilation units y such that y directly dependson x
```

```
    loop
      invalidate (y)
    end loop;
  end;
```

where

```
  invalidate (x)
    if is_invalid (x)
    then
      return;
    end if;
    forall compilation units y such that y directly depends on x
    loop
      invalidate (y);
    end loop;
    mark x invalid
  end;
```

The algorithm recursively initiates the compilation of all units that are necessary and sufficient for the compilation of a particular unit. After the compilation of a unit has succeeded and the result of the compilation has been entered in the program library all units previously depending on the unit so compiled are invalidated. (An implicit assumption is that all units in the set are marked either *valid* or *invalid*.)

Modifying a unit and adding a compiled version to the library often leads to invalidating a large number of units because of the various and possibly unexpected dependencies among units. Since most modifications to existing systems have only a local effect, it can be argued that in those cases no recompilations, or at least fewer recompilations, are required.

Algorithms - in an environment of another language - for the reduction of the number of units to be recompiled are given by Rain [Rain-84]. These algorithms are based on the observation that the effect of a change is usually local, such that:

1. a unit that depends on another unit, the latter being affected but its compilation remaining unchanged, does not have to be recompiled. The change then is likely to have been of a lexical nature, probably in the spirit of adding or removing a comment or of changing the program layout.
2. when a unit *is* affected, but the compiled versions of the units depending directly on it remain unchanged then the units depending on the latter do not need to be recompiled;

This process can be generalized. If, in the process of recompilation of all units directly and indirectly depending on an affected unit, it turns out that the compiled version of a particular unit remains unchanged, then there is no need for the recompilation of units

depending on the latter.

A simple example should suffice, however, to demonstrate that such an algorithm should be handled with the greatest possible care in an environment of the Ada language.

```
package a is
  aa : integer;
end;

with a; use a;
package b is
  cc : integer renames aa;
  package dd renames a;
end;
```

After the addition of a declaration

```
ee : integer;
```

to the declarative part of package *a*, the compiled version of package *b* is not necessarily changed (at least in the Ada— compiler the intermediate program representation is the same). This would indicate that the units that directly depend on *b* do not have to be recompiled. Consider by contrast the example of a package *ff*, given below:

```
with b; use b;
package ff is
  use dd;
  ...
end;
```

The change in the package specification *a* has a definite effect on *ff*. Despite the fact the the compiled version of *b* remains unchanged, a compiled version of *ff* should be made invalid.

The elaboration order of compilation units

A complete program consists of a main program, and any such library units and their subunits and bodies mentioned in *with clauses*; this rule is to be read recursively. The elaboration of these units is performed consistently with the dependence relations. Furthermore, the elaboration of package bodies must be consistent with any dependence relations resulting from the actions performed during the elaboration of these bodies. The intent of the rules establishing an elaboration order is to ensure that a declaration is always established before it is used.

The elaboration order of the components that form a complete program is only partially defined; in general more than a single elaboration order, complying with the rules given above, can be computed. Yet an order of elaboration consistent with the rules above is not necessarily sufficient to ensure that each library unit is elaborated before any other compilation unit, whose elaboration, as it happens, necessitates the library-unit body to be elaborated earlier. Consider, by way of example, the package specifications *p* and *q* and

their bodies as given below:

```
package p is
   ...
end;

package q is
   ...
end;
```

and the package bodies:

```
with q;
package body p is
   ...
end;

with p;
package body q is
   ...
end;
```

Except for the bodies *p* and *q* being strictly local and non-overlapping in some sense, no elaboration ordering is possible here. The body of *p* uses elements of *q*. Therefore, it seems reasonable to ensure that the body of *q* is elaborated prior to the elaboration of *p*. Unfortunately, the same reasoning applies to the body of *q*. To provide for such cases, a pragma is defined† (section 10.5 of [LRM-83]) to prompt a compiler with additional information on the elaboration order desired.

7.4 Program libraries for the Ada— Compiler

During the development of the Ada— compiler several models of program libraries were investigated and implemented. In this section two of these models are briefly discussed. The section is concluded by a short discussion of current developments in the area of program libraries for the Ada— compiler.

7.4.1 A simple closed model

The first library used in conjunction with the DAS compiler was based on some ideas from Dausmann et al [Dausmann-80a] and from Hans van Someren [Someren-82b]. The program library was implemented as a *closed system* with which the user interacted by a small invariable set of commands. Commands were available to:

- compile and to enter a source in the library;
- extract a source from the library;
- compile all units required for the compilation of a given unit;

---

† this pragma is not yet implemented in the Ada— compiler.

- load a unit, i.e. to make an executable.

The implementation of this library was straightforward. Central in the program-library implementation was a *masterfile*. This masterfile acted as a *library descriptor*; it contained a description of all library units. Library units were identified by an integral value in the range 1 .. 65535 †. The implicit assumption in this library model is that the description in the masterfile is always consistent with the actual state of the program library.

At the time this program library model was designed and implemented, DAS was only a small subset of the Ada language. Notably, the lack of features such as e.g. separate subunits simplified the design and implementation of the program-library manager.

The program library based on this model worked quite satisfactorily for some time. A drawback from an implementor's point of view was that the programs implementing the library manager assumed the description of the library to be consistent with its actual state. No guarantee of this assumption could be given. Whenever a user accessed and modified a library element with ordinary UNIX tools, e.g. the editor or a copy command, the description of the program library in the master file became potentially inconsistent with the actual state of the program library, which was discouraged but could not be prohibited.

A drawback from the user's point of view was that legal manipulation of compilation unit was markedly different from the manipulation of other entities in the UNIX operating system. It was felt that an approach closer to a UNIX-style was needed.

7.4.2 A UNIX-style approach

A significant characteristic of any UNIX-style tool is that its domain is formed by UNIX files. Consequently, a characteristic of a UNIX-style program-library manager is that the user is led into thinking he manipulates UNIX files.

Having experimented with several intermediate forms of more-or-less closed program libraries, experiments were performed with a UNIX-style program library. In this model the user was led into thinking he manipulates ordinary UNIX files whenever possible. For actions such as e.g. the compilation of a unit, the user should not even be aware of the fact that most commands operate on a program library. The basic idea was that data kept in a library descriptor should *assist* the user in performing his work whenever reasonably possible. If, however, support should turn out to be impossible or too expensive to implement, it should be withheld.

For purely technical reasons it was decided to drop the demand that a library descriptor should reflect at all times the actual state of the corresponding program library. It was accepted that a library descriptor could become inconsistent with that library's contents. Since this inconsistency was anticipated and descriptive data was kept in a program-library descriptor file, it was always possible to regenerate a consistent view of the library.

---

† These numbers, Compilation Unit Numbers or *cuns* for short, were not only used for the library manager. The compiler's back end also uses these numbers to generate unique names for data items in the various compilation units.

In this approach the notion of *masterfile* remains intact. A masterfile contains a description for every unit compiled. It maintains for each unit included:

1. the name of the source file;
2. the name of a file containing the intermediate program representation. The intermediate program representation kept in the file contains only declarative information potentially to be used by other compilation units;
3. the name of a file containing the object;
4. the list of units on which this unit *directly depends* (the *with list*);
5. the list of subunits and separate units belonging to this compiled unit;
6. (when applicable) the name of the executable program.

When starting with the compilation of a compilation unit no assumption whatsoever is made on the consistency of the library description. The library description is used as a guide to check consistency. The library manager dynamically checks the validity of the units to be used for the compilation. As soon as a compilation unit is entered into the program library, all compilation units on which this compilation depends by construction are already in the library.

Now assume the user modifies a source file that contains the source of a package *P*. Recompilation of this source will cause any element in the library depending on package *P* to become invalid. These elements are not marked invalid, however. Instead, two other actions are taken:

1. whenever a request is made for a compilation unit, a validity check is performed on the fly. This check will point out the invalidity of any library unit to be used if it either has been changed or depends on a unit changed.
2. in all normal cases, the library-description file contains sufficient data to automatically recompile any unit noted as invalid in step 1.

Obviously, if one has removed or renamed source files, the data in the library descriptor file does not reflect this and no automatic recompilation is then possible. Indeed, automatic recompilation is possible under the assumption that the relation between the source file name and the corresponding library units is invariant.

The actual library structure is somewhat more complex than suggested here, primarily because of the need to take secondary units and subunits into account.

The program library contains library units. A library-unit descriptor may contain a reference (references) to descriptors for its secondary units. Descriptors for secondary units may contain references to descriptors for their subunits, as depicted in the following:

```
package a is
   ...
end;
```

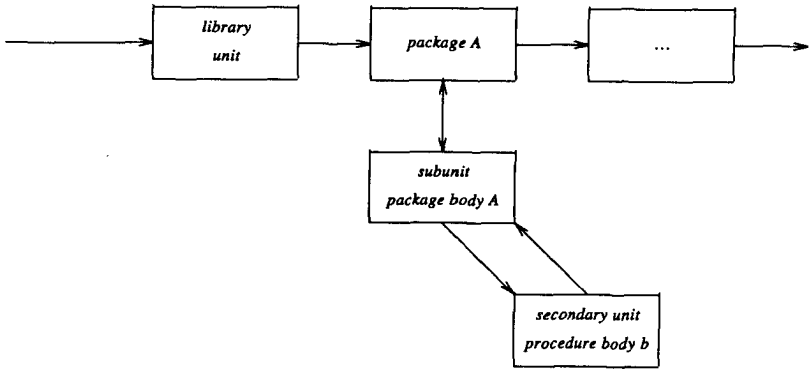
The body is a secondary unit:

```
package body a is
  ...
  procedure b is separate;
  ...
end;
```

The structure of the subunit is schematically

```
separate (a)
procedure b is
  ...
end;
```

The structure of the library descriptors and descriptors for subunits is schematically given below.



It may be obvious from the description that implementation techniques as given here can only be used with small-to-medium scale program libraries. For libraries containing thousands and thousands of interdependent modules, completely different implementation paradigms may have to be used.

7.4.3 Library maintenance tools

Keeping the library up to date is accomplished either by hand or by using some special recompilation and maintenance tools. The possibility of automatic recompilation involves the retaining of some additional data since the data kept in the library-description file is not completely sufficient to perform these recompilations automatically.

Tools can be built to extract data from the library description file and to use this data to perform some operations. In particular tools were designed for:

- listing the library contents;
- patching the library;
- recompiling units that had become obsolete and that could not be automatically recompiled by the mechanism described above.

In order to take care of the computation of the compilation order in a set of source files, other tools were designed. A *library-update tool,* loosely based on Adadep ([Briggs-84]), was designed and built [Biegstraaten-87]. This tool takes as its argument a set of Ada language source files and (optionally) a library-descriptor file. It either generates a sequence of commands to (re)compile the source files in the context of the library or it initiates the compilation itself. As an option, it shows the structure of the programs and the interdependence between the contained Ada compilation units.

Roughly speaking, the tool consists of a common front end and back end. The front end essentially is a simple parser, based on a context-free grammar for a superset of the Ada language. The back end interprets the output of the front end, it consults, if required, the library-descriptor file and it generates, at will:

- a standard UNIX *make* file to recompile any subset of the set of source files comprising (a part of) an Ada program library;
- a sequence of commands performing the recompilation of any subset of the set of source files comprising an Ada program library;
- a formatted output list of all dependency relations;
- a formatted output list of the program structure of the individual compilation units.

Depending on the availability of data in a program-library descriptor file, the tool is able to specify names of output files as they might have been after an immediately preceding compilation. If a file is compiled for the first time (or for any other reason no description of the file is found in a library description file) default names are used as names for the output files.

The tool can be used in an environment where other library models or other library implementations are supported. Provided a set of files is closed with respect to interdependency with other compilation units, the library update tool generates a complete compilation script.

Compiler and loader as library tools

The Ada compiler and the Ada loader are probably the most important users of the library.

The compiler reads a source file and translates each compilation unit of the source file into an object file and, in case declarative information of the unit has to be preserved, a tree structure. Successful compilation is a prerequisite for a compilation unit to be entered into the library. The output of the compiler consists, apart from the usual output in terms of binary and error files, of a small description. This description contains the encoded information for each compilation unit processed in the current compilation. It consists of

a single line of ASCII code in which the relevant characteristics of the processed compilation unit are encoded.

The Ada loader takes as its argument the name of the body to be loaded. It determines the units that must be included in the load module and it computes an elaboration order (recall that the Ada loader generates code to perform the elaboration of the library units, chapter 5.6.3) and it calls the system link editor.

7.4.4 Current developments

It is well realized that the current program-library structure is ill-suited as a vehicle for the support of large-scale software development. In particular for the latter to be feasible, there must be prior solutions to problems such as:

- source-code-version management;
- sharing of compilation units among libraries.

Source-code management is an important topic. Recently, several papers were published on the subject in conjunction with Ada program libraries [Dausmann-87], [Narfelt-84], [Narfelt-85], [Schefstrom-86], and [Wilson-86]. At Delft University of Technology a more advanced program-library management system is being built [Dusink-87]. This management system aims at integrating the library manager of the Ada— compiler with the UNIX environment. It will allow:

- sharing of compilation units among different libraries,
- the existence of several versions of the same compilation unit in a single program library.

8. Results and conclusions

In this thesis we have described some elements in the design and implementation of the Ada— compiler, a compiler for a very large subset of the Ada language.

The objectives of this research were twofold. First, surveying the literature related to the implementation of the Ada language; second, developing and disclosing technology and algorithms in the form of a prototype implementation. These objectives were reached. Seen as an educational excercise the implementation was a success. Implementing a language of this size does not only teach about building compilers and developing compiler technology, it also teaches some object lessons in software engineering.

The Ada language is a large language; compiling it requires a large compiler. Even the Ada— compiler, a compiler that does not implement the full Ada language, is quite large. It is about ten times the size of an average Pascal-to-P-code translator. With the current compiler technology and current hardware, this kind of compilers is slow. The current compilation speed of our compiler is about 20 lines/second on an M68020 based system. On the other hand, the algorithms required for the compilation of the Ada language are known; their implementation can be profiled and tuned. It seems perfectly possible to increase the efficiency of the whole compiler by a significant percentage. By carefully tuning the code, the performance should be improved by 20 to 30 %. A performance improvement of the same order seems possible by choosing other algorithms. An overall performance improvement of about 50 % seems realistic.

The general implementation strategy applied in the compiler turns out to be satisfactory. Separating the front end and the back end keeps both parts within manageable size and complexity. The option taken, viz. to maintain a complete intermediate program representation in memory was also a favourable decision; the manipulation of any complete tree structure is simpler than moving back and forth over a linearized external program representation, both conceptually and from an implementer's viewpoint. A consequence of the strategy applied is that that a compilation requires a large chunk of memory. Since it is to be expected that the price of memory (and the price and price/performance ratio of CPU's) will continue to decrease, the choice seems justifiable. The availability of large amounts of memory to the compiler user allows essentially different paradigms from the paradigms required for compiler development and usage on a 64 Kb PDP-11.

The front end of the compiler was built using fairly conventional technology. Parser and scanner were built with generators, most of the remaining part was hand crafted. Our initial idea was to use a formal definition of the Ada language as a prototype compiler description which was then to be translated by hand. Our experience was that the available formal definitions were incomplete and also contained numerous errors. Using it as a basis for direct translation by hand was agonizing. Nevertheless, we tried in the further stages of our front-end design to remain as true as possible to this formal definition. The main efforts in the design and implementation of the compiler front end were in experimenting with algorithms and implementation techniques in the areas of overload resolution (taking derived and universal types into account) (subsection 4.5.2), and in declaration processing

and identifier look-up (subsection 4.5.3). We believe that in particular the approach taken in the implementation of the scope and visibility rules is a sound one. It is a suitable candidate for further formalization and may serve as a basis for the mechanical construction of efficient symbol-table packages. As it turned out, the implementation of generics (section 4.6), using the macro model and imposing a few restrictions, was simpler than expected. A drawback of the method that was applied is the duplication of code for the instantiations.

Designing and implementing the back end of the compiler was a completely new experience. Literature on compiler back ends is scarce, literature on efficient implementations even more. At a very early stage, it was decided that a code generator should be borrowed which defined the output language of the expander. The choice to design and to implement our own run-time data description model (section 5.3) turned out to be favourable. The model is simpler than the alternative dope-vector-based model. Furthermore, it is conceptually simple and the generation of production-quality code is relatively easy. The algorithms applied to implement the model are almost free of dependencies on both the source language and the target architecture, while still generating production-quality code. It seems that the model and the implementing algorithms are good candidates for further formalization.

Tasking remains an intriguing area. Although only a prototype of a tasking supervisor was built (chapter 6), this prototype was as fast (or as slow) as other tasking kernels that were reported in the literature. Further scrutiny indicated that the construction of an *efficient* implementation of Ada tasking is a major undertaking. It is worthwhile to tune the current implementation and to investigate the relation between the implementation strategy and the resulting real-time performance. A second point for further research is the investigation of tailored implementations, i.e. the implementation of only a subset of the Ada tasking semantics dedicated to a certain application or application domain, and the impact of such subsetting on the real-time capabilities of the tasking kernel.

Finally, separate compilation is also a topic that deserves further research (chapter 7). The impact of a program library on the speed with which a compilation can be performed is large. For smaller systems that must be compiled the compilation time may remain determined by the speed of the compiler. For larger systems that must be compiled, the compilation speed is completely limited by the need to read a large number of intermediate representations. Reading intermediate-program representations as our implementation does, is certainly a time-consuming process; for programs referring (either explicitly or implicitly) to a large number of library units, better solution may have to be found.

Concluding, we may say that the design and implementation of the Ada— compiler has reached its two major objectives. The algorithms are presented, in order to promote the public knowledge on Ada language implementations.

9. References

[ACE-85]
The intermediate code, user document,
ACE, Associated Computer Experts,
Amsterdam, 1985.

[AIE-82]
Computer Program Development Specification for Ada Integrated Environment: Ada Compiler Phases,
Intermetrics Cambridge, 1982.

[Ada-79]
Preliminary Ada,
Sigplan Notices 14 (6), June, 1979.

[Ada-80]
Reference Manual for the Ada Programming Language,
U.S. Dept of Defense, November, 1980.

[Aho-85]
Compilers: principles, tools and techniques,
A. Aho, R. Sethi, J. Ullman,
Addison Wesley, 1985.

[ALGOL-68]
Revised Report on the Algorithmic Language Algol 68,
A van Wijngaarden et al.
MC Tract 50, Mathematical Centre Amsterdam, 1976.

[Appelbe-82]
An operational definition of intermediate code for implementing a portable Ada compiler,
B. Appelbe, G. Dismukes,
in: Proceedings of the AdaTec conference on Ada, Arlington October, 1982 pp 266 - 274.

[Ardo-84]
Experimental implementation of an Ada tasking run-time system on the multiprocessor computer C_m^*,
A. Ardo,
in: Proceedings of Washington Ada symposium, 1984 ACM, 1984 pp 145 - 153.

[Baker-82a]
A single-pass syntax directed front-end for Ada,
T.P. Baker,
in: Proceedings Sigplan Symposium on Compiler Construction, 1982
Sigplan Notices 17 (6) June, 1982 pp 318 - 326.

[Baker-82b]
A One-pass algorithm for overload resolution in Ada,
T.P. Baker,
ACM Transactions on Programming Languages and Systems 4 (4) October, 1982 pp 601 - 614.

[Baker-84]
A Run-time Supervisor to Support Ada Task Activation, Execution, and Termination,
T.P. Baker, G. A. Ricardi,
in: IEEE Computer Society, 1984 Conference on Ada Applications and Environments IEEE, 1984 pp 14 - 22.

[Baker-86a]
An Ada Runtime System Interface,
T.P. Baker,
IEEE second international conference on Ada Applications and Environments pp 99 - 114. 1986.

[Baker-86b]
Ada tasking: From semantics to efficient implementation,
T.P. Baker, G.A. Riccardi,
IEEE Software 2 (2) March, 1985 pp 34 - 46.

[Baker-86c]
Implementing Ada Exceptions,
T.P. Baker, G.A. Riccardi,
IEEE Software 3 (5) September, 1986 pp 42 - 51. 1986.

[Bal-82]
Overloading resolution in DAS,
H.E. Bal,
Unpublished Masters Thesis, Department of Mathematics and Computer Science Delft University of Technology, 1982.

[Barbacci-85]
The Ada+ compiler front end and Code generator,
M. Barbacci, W.H. Maddox, T.D. Newton, R.G. Stockton,
Proceedings of the Ada international Conference, 1985 Ada in Use,
Cambridge University Press, 1985 pp 343 - 354.

[Barkey-83]
Using PCC's backend for code generation in the DAS compiler,
J.A. Barkey,
Unpublished Masters Thesis, Department of Mathematics and Computer Science Delft University of Technology, 1983.

[Barkey-85]
Using 'off the shelf' code generators in the DAS compiler: some experiences,

C. Barkey, J. van Katwijk, W.J. Toetenel,
Report 85-37 Department of Mathematics and Computer Science
Delft University of Technology, 1985.

[Bauner-80]
An implementation and evaluation of the real-time primitives in the programming language Ada,
J.-O Bauner, G. Svensson,
Report TRITA-CS-8001
Department of Telecommunication and Computer Systems
The Royal Institute of Technology, Stockholm, Sweden, 1980.

[Bayan-86a]
Requirements for a real-time Ada run-time kernel and proposed interface,
R. Bayan, C. Bonnet, A. Kung, W. Kirchgassner, R. Landwehr, B. Schwarz,
Project Asterix
TECSI software, Paris 1986.

[Bayan-86b]
Transition toward embedded real-time applications in Ada,
R. Bayan, C. Bonnet, A. Kung, W. Kirchgassner, R. Landwehr, B. Schwarz,
in: Proceedings of the Ada-Europe international conference on Ada, 1986 Managing the Transition,
Cambridge University Press, 1986 pp 153 - 164.

[Belmont-80]
Type resolution in Ada: An implementation Report,
P.A. Belmont,
in: Proceedings of the ACM-Sigplan symposium on the Ada Programming Language
Sigplan Notices 15 (11) november, 1980 pp 57 - 61.

[Belz-80]
A Multiprocessing Implementation-Oriented Formal Description of Ada in Semanol,
F.C. Belz, E.K. Blum, D. Heimbigner,
in: Proceedings of the ACM-Sigplan Symposium on the Ada Programming Language
Sigplan Notices 15 (11) November, 1980 pp 202 - 212.

[Biegstraaten-87]
Ada— user's manual,
A.W.W.M. Biegstraaten, J van Katwijk, W.J. Toetenel,
Report 87-XX, Faculty of Mathematics and Computer Science,
Delft University of Technology (To appear November, 1987).

[Bishop-80]
Effective Machine Descriptors for Ada

J.M. Bishop,
in: Proceedings of the ACM-Sigplan symposium on the Ada Programming Language,
Sigplan Notices 15 (11) November, 1980 pp 235 - 242.

[Blower-84]
Efficient implementation of visibility in Ada,
M.I. Blower,
in: Proceedings of the Sigplan' 84 symposium on Compiler Construction
Sigplan Notices, 19 (6) June 1984 pp 259 - 265.

[Bogstad-83]
An interpreter for DAS,
N.C. Bogstad, A.M.J. Hartveld,
Report 51560-28(1983)16
Department of Electrical Engineering
Delft University of Technology, 1983.

[Bondeli-83]
Models for the control of concurrency in Ada based on predicate-transition nets,
P. de Bondeli,
Ada-Europe/AdaTec Joint Conference on Ada
Brussels March, 1983

[Bonet-81]
Ada syntax diagrams for top-down analysis,
R. Bonet. A. Kung, K. Ripken, R.K. Yates, M. Sommer, J. Winkler,
Sigplan Notices 16 (9) September, 1981 pp 29 - 41.

[Boom-86]
A view of formal semantics,
H.J. Boom (editor),
Guidelines for the preparation of standards within SC22
ISO/TC97/SC22 Document N247, 1986.

[Bray-83]
Implementation implications of Ada generics,
G. Bray,
Ada Letters 3 (2) September/October, 1983 pp 62 - 71.

[BreadBoard-82]
The Ada Breadboard compiler,
Crowley, Langer, Lettvin, Nowitz, Quinn, Rubine, Whetherell,
AT&T Laboratories 1982.

[Briggs-83a]
Ada Workbench Compiler Project, 1982,
J.S. Briggs et al,
University of York YCS 59, 1983.

[Briggs-83b]
The design of AIR and its application to Ada separate compilation,
J.S. Briggs,
in: Ada Software Tools Interfaces, Edited by P.J. Wallis
Springer Verlag LNC 180, 1983.

[Briggs-84]
Two implementations of the Ada Program Library,
J.S. Briggs,
Software - Practice and Experience 15 (5) May, 1985 pp 491 - 500.

[Bruun-81]
Portable Ada Programming System, Ada Static semantics, Well-formedness criteria,
H. Bruun, J. Bundgaard, J. Jörgensen,
Dansk Datamatik Center 1981.

[Bundgaard-82]
Portable Ada Programming System: Test Front End Compiler,
J. Bundgaard, J. Jörgensen, H. Bruun,
Global Design
Dansk Datamatik Center, 1982.

[Bundgaard-85]
The development of an Ada Front End for Small Computers,
J. Bundgaard,
in: Proceedings of the Ada International Conference, 1985 Ada in Use,
Cambridge University Press, 1985 pp 321 - 328.

[Burger-87]
An assessment of the overhead associated with tasking facilities and task paradigms in Ada,
T.M. Burger, K.W. Nielsen,
Ada Letters 7 (1) January/February, 1987 pp 49 - 58.

[Burns-85]
A review of Ada tasking,
A. Burns, A.M. Lister, A.J. Wellings,
University of York, Dept of Computer Science YCS-78, 1985.

[Carney-86]
Software Methodology and the AIE Program Library,
D. Carney,
In: Proceedings of the Ada international Conference, 1986 Managing the Transition,
Cambridge University Press, 1986 pp 115 - 124.

[Chow-83]
Intermediate languages in compiler construction - a bibliography,
F.C. Chow, M. Ganapathi,

Sigplan Notices 18 (11) November, 1983 pp 21 - 23.

[Clemmensen-82]
A formal model of distributed Ada tasking,
G.B. Clemmensen,
in: Proceedings of the AdaTec conference on Ada Arlington, October, 1982 pp 224 - 237.

[Clemmensen-83]
Portable Ada Programming System: back end compiler,
Specification of the language *IML7*
G.B. Clemmensen, J.S. Pedersen,
Document DDC 02/RPT/37
Dansk Datamatik Center, 1983.

[Cole-81]
Ada Syntax Cross Reference,
S.N. Cole,
Sigplan Notices 16 (3) March, 1981 pp 18 - 47.

[Conradi-85]
Mechanisms and Tools for Separate Compilation,
Draft Version
R. Conradi, D. H. Wanvik,
University of Trondheim, Technical report 85/25, 1985.

[Cormack-81]
An algorithm for the selection of overloaded functions in Ada,
G.V. Cormack,
Sigplan Notices 16 (2) February, 1981 pp 48 - 52.

[Cornhill-83]
A survivable distributed computing system for embedded applications written in Ada,
D.T. Cornhill,
Ada Letters 3 (3) November/December, 1983 pp 79 - 87.

[Cornhill-84]
Four approaches to partitioning Ada programs for execution on distributed targets,
D.T. Cornhill,
In: IEEE computer society, 1984 Conference on Ada Applications and Environments IEEE, 1984 pp 153 - 162.

[DIANA-83]
DIANA Reference Manual,
G. Goos, W. A. Wulf,
Revision 3 by:
A. Evans, K. J. Butler,
Tartan Laboratories Inc. Pittsburg 1983.

[Dausmann-80a]
SEPAREE: A Separate Compilation System for Ada,
M. Dausmann, S. Drossopoulou, G. Persch, G. Winterstein,
Bericht 32/80 Nov. 80 Fakultät für Informatik, Universität Karlsruhe.

[Dausman-80b]
Efficient Recompilation Checks for Ada,
M. Dausmann, S. Drossoupoulou, G. Persch, G. Winterstein,
Bericht 30/80 Nov. 80 Fakultät für Informatik, Universität Karlsruhe.

[Dausmann-80c]
On Reusing Units of Other Program Libraries,
M. Dausmann, S. Drossoupoulou, G. Persch, G. Winterstein,
Bericht 31/80 Nov. 80 Fakultät für Informatik, Universität Karlsruhe.

[Dausmann-87]
Version control and separate compilation in Ada,
M. Dausmann,
in: Proceedings of the Ada-Europe International conference, Stockholm, 1987
Cambridge University Press, 1987 pp 157 - 169.

[Dewar-80]
The NYU Ada translator and interpreter,
R. Dewar, G. Fisher, E. Schonberg R. Froehlich. S. Bryant C. Goss, M. Burke,
in: Proceedings of the Sigplan Symposium on the Ada Programming Language
Sigplan Notices 15 (11) November, 1980 pp, 194 - 201.

[Dewar-83]
Executable Semantic Model for Ada,
Ada/Ed interpreter
R.B.K. Dewar, R. M. Frochlich, G.A. Fisher, Ph. Kruchten,
Ada Project Courant Institute, New York University, 1983.

[Dismukes-84]
Private Communication.

[Dross-82]
An attribute Grammar for Ada,
S. Drossoupoulou, J. Uhl, G. Persch, G. Goos, M. Dausmann, G. Winterstein,
in: Proceedings of the Sigplan Symposium on Compiler Construction, 1982
Sigplan Notices 16 (6) June, 1982 pp 334 - 349.

[Dross-83]
Measurements of an Ada compiler,
Sophia Drossoupoulou et al.
Ada-Europe/AdaTec Conference on Ada
Brussels March, 1983.

[Dusink-87]
An implementation of an advanced Ada program library,
E.M. Dusink,
Report 87-XX, Faculty of Mathematics and Computer Science,
Delft University of Technology (To appear October, 1987).

[Dijkstra-83]
A design of a runtime kernel for the implementation of Ada multitasking,
K. Dijkstra, M.R. Groothuizen,
Unpublished Report, Department of Mathematics and Computer Science
Delft University of Technology, 1983.

[Dijkstra-84]
A debugger for DAS,
K. Dijkstra,
Unpublished Masters Thesis, Department of Mathematics and Computer Science
Delft University of Technology, 1984.

[Falis-82]
Design and implementation in Ada of a runtime task supervisor,
E. Falis
in: Proceedings of the AdaTec conference on Ada Arlington, October, 1982 pp 1 - 9.

[Feldman-79a]
Implementation of a portable Fortran 77 compiler using modern tools,
S.I. Feldman,
in: Proceedings of the Sigplan Compiler Symposium on Compiler Construction 1979
Sigplan Notices 14 (8) August, 1979 pp 98 - 106.

[Feldman-79b]
Make - a program for maintaining computer programs,
S.I. Feldman
Software - Practice and Experience 9 (4) April, 1979 pp 255 - 265.

[Fisher-84]
A LALR (1) grammar for ANSI Ada,
G. Fisher, P. Charles,
Ada Letters 3 (4) January/February, 1984 pp 37 - 50.

[FormDef-80]
Formal Definition of the Ada Programming Language,
CII Honeywell Bull,
Louveciennes, France, Preliminary version for Public Review 1980.

[FormDef-86]
The Draft Formal Definition of Ada,
Static Semantics of an Example Ada Subset
J. S. Pedersen,

Dansk Datamatik Center, 1986.

[Ganzinger-80]
Operator identification in Ada,
H. Ganzinger, K. Ripken,
Sigplan Notices 15 (9) September, 1980 pp 30 - 43.

[Garlington-81]
Preliminary design and implementation of an Ada Pseudo-machine,
A.R. Garlington,
Msc Thesis, School of Engineering Air Force Institute of Technology
Wright-Patterson Air Force Base Ohio, 1981.

[Goor-81]
A higher-level language machine,
A.J. van de Goor,
Unpublished Memorandum Department of Electrical Engineering Delft University of Technology, 1981.

[Goos-80]
Towards a compiler front end for Ada,
G. Goos, G. Winterstein,
in: Proceedings of the Sigplan Symposium on the Ada Programming Language
Sigplan Notices 15 (11) November, 1980 pp 36 - 46.

[Goos-83]
Compiler Construction,
G. Goos, W. Waite,
Springer Verlag, 1983.

[Graham-79a]
Hashed Symbol Tables for Languages with Explicit Scope Control,
S.L. Graham, W.N. Joy, O. Roubine,
in: Proceedings of the Sigplan Symposium on Compiler Construction, 1979
Sigplan Notices 14 (8) August, 1979 pp 50 - 57.

[Graham-79b]
Practical LR Error Recovery,
S.L. Graham, W.N. Joy,
in: Proceedings of the Sigplan Symposium on Compiler Construction, 1979
Sigplan Notices 14 (8) August, 1979 pp 168 - 175.

[Gries-71]
Compiler Construction for Digital Computers,
D. Gries,
J. Wiley, 1971.

[Griswold-77]
An overview of SL5,

R.E. Griswold, D.R. Hanson,
Sigplan Notices 12 (1) January, 1977 pp 40 - 50.

[Groeneveld-86]
VEYACC, some extensions to YACC,
M. Groeneveld,
Unpublished Masters Thesis, Department of Mathematics and Computer Science
Delft University of Technology, 1986.

[Grovers-80]
The design of a virtual machine for Ada,
L.J. Grovers, W.J. Rogers,
in: Proceedings of the ACM-Sigplan symposium on the Ada programming language
Sigplan Notices 15 (11) November, 1980 pp 223 - 234.

[Gupta-85]
The efficiency of storage management schemes for Ada programs,
R. Gupta, M.L. Soffa,
Sigplan Notices 20 (11) November, 1985 pp 30 - 38.

[Haridi-81]
An implementation and empirical evaluation of the tasking facilities in Ada,
S. Haridi, J.-O Bauner, G. Svensson,
Sigplan Notices 16 (2) February, 1981 pp 35 - 47.

[Hartig-81]
Task state transitions in Ada,
H. Hartig, A. Pfitzmann, L. Treff,
Ada Letters 1 (1), July/August, 1981 pp 31 - 42.

[Heliard-82]
The European Ada compiler Project,
J.C. Heliard, O. Roubine, J. Teller,
Joint Ada-Europe/Ada-TEC Meeting
European Commission, Brussels March 1892.

[Heliard-83]
Compiling Ada,
J.C. Heliard,
in: Methods and Tools for Compiler Construction
Edited by B. Lorho
Cambridge University Press, 1984

[Hilfinger-82]
Implementation strategies for Ada Tasking Idioms,
P.N. Hilfinger,
in: Proceedings of the AdaTec conference on Ada Arlington October, 1982 pp 10 - 25.

[Hisgen-80]
A run-time representation for Ada variables and types,
A. Hisgen, D.A. Lamb, J. Rosenberg, M. Sherman,
in: Proceedings of the ACM-Sigplan symposium on the Ada programming language
Sigplan Notices 15 (11) November, 1980 pp 82 - 90.

[Hoare-78]
Communicating sequential processes,
C.A.R. Hoare,
Communications of the ACM 21 (8) August, 1978 pp 666 - 677.

[Huijsman-87]
Performance aspects of Ada tasking in Embedded systems,
R.D. Huijsman, J. van Katwijk, W.J. Toetenel,
Paper accepted for Euromicro-87.

[Ibsen-82]
A-machine specification,
L. Ibsen, L.O.K. Nielsen, N.M. Jörgensen,
Christian Rovsing A/S Copenhagen Denmark, 1982.

[Irvine-83]
User's manual for ICSC-Ada V2.2,
Irvine Computer Science Corporation,
August, 1983.

[Janas-80]
A comment on "Operator Identification in Ada",
J.M. Janas,
Sigplan Notices 15 (9), September, 1980 pp 39-43.

[Jansohn-82]
A code generator for Ada,
H.S. Jahnson, R. Landwehr,
Bericht 33/82 Fakultät für Informatik
Universität Karlsruhe, 1982.

[Johnson-74]
YACC, Yet another Compiler Compiler,
S.C. Johnson,
Bell Labs Murray Hill New Yersey, 1974.

[Johnson-78]
A portable compiler: Theory and practice,
S.C. Johnson,
in: Fifth Annual ACM Symposium on Principles of Programming Languages, 1978,
pp 97 - 104.

[Johnson-81]
The C Language Calling Sequence,
S.C. Johnson, D.M. Ritchie,
Computing Science Technical report 102, AT &T Bell Laboratories, 1981.

[Johnson-82]
Semantic errors, diagnosis and repair,
C.W. Johnson, C. Runciman,
in: Proceedings of the Sigplan Symposium on Compiler Construction, 1982
Sigplan Notices 17 (6) June, 1982 pp 88 - 97.

[Jones-82]
Comparative efficiency of different implementations of the Ada Rendezvous,
A. Jones, A. Ardo,
in: Proceedings of the AdaTec conference on Ada Arlington October, 1982 pp 212 - 223.

[Joy-83]
Berkeley Pascal User's Manual, Version 3.0 W.N. Joy, S.L. Graham, C.B. Haley, M.K. McKusick, P.B. Kessler,
in: UNIX-4.2 BSD documentation.

[Kamel-84]
Further Experiences with Separate Compilation at BNR,
R.F. Kamel, N.D. Gammage,
in: Preprints Proceedings TC2 Working Conference Systems Implementation Languages, Experience and Assessment September 1984.

[Kamrad-83]
Runtime organization for the Ada Language System Programs,
J.M. Kamrad III,
Ada Letters 3 (3) November/December, 1983 pp 58 - 68.

[Kastens-83]
GAG, A Generator based on attributed grammars,
U. Kastens, E. Zimmermann,
Springer Verlag, 1983.

[Katwijk-83]
A preprocessor for YACC or a poor man's approach to parsing attributed grammars,
J. van Katwijk,
Sigplan Notices 18 (10) October, 1983 pp 12 - 15.

[Katwijk-84a]
The doublet model,
J. van Katwijk, J. van Someren,
Sigplan Notices 19 (1) January, 1984 pp 78 - 92.

[Katwijk-84b]
Practical Experiences with Automatic Repair of Syntactical Errors or Syntactical Error Repair in the DAS compiler,
J. van Katwijk,
Sigplan Notices 19 (9) September, 1984 pp 37 - 48.

[Katwijk-87a]
Addressing types and objects in Ada,
J. van Katwijk,
Software - Practice and Experience 17 (5) May, 1987 pp 319 - 343.

[Katwijk-87b]
An Ada tasking supervisor; description of an implementation,
J. van Katwijk, W.J. Toetenel,
Report 87- XX, Faculty of Mathematics and Computer Science,
Delft University of Technology, 1987 (To appear, November, 1987).

[Kirchgasner-82]
An optimizing Ada compiler,
W. Kirchgasner, J. Uhl, G. Persch. G. Winterstein, G. Goos, M. Dausmann, S. Drossoupoulou,
Bericht 29/82 Fakultät fur Informatik,
Universität Karlsruhe, 1982.

[Konijn-84]
Retargeting the DAS compiler with the EM Machine Model,
E. van Konijnenburg,
Unpublished Masters Thesis, Department of Mathematics and Computer Science
Delft University of Technology, 1984.

[Kruchten-86]
Une Machine Ada Virtuelle: Architecture,
Ph. Kruchten,
These pour le Doctorat de l'ENST
Ecole Nationale Superieure des Telecommunications Paris 1986.

[LRM-83]
Reference Manual for the Ada Programming Language,
MIL-STD 1815-a, 1983.

[Lahtinen-82]
A machine architecture for Ada,
P. Lahtinen,
Ada Letters 2 (2) September/October, 1982 pp 28 - 33.

[Leathrum-84]
Design of an Ada run-time system,
J.F. Leathrum,
in: IEEE Computer society, 1984 Conference on Ada Application and

Environments IEEE, 1984 pp 4 - 13.

[Lesk-75]
Lex, a lexical analyzer generator,
M. Lesk,
Bell Labs Murray Hill New Yersey, 1975.

[Lettvin-82]
The Ada BreadBoard Compiler: The Library and Dictionary,
J.D. Lettvin,
In : [BreadBoard-82].

[Liere-87]
Three implementations of the DAPSE,
R. van Liere,
Unpublished Masters Thesis, Department of Mathematics and Computer Science
Delft University of Technology, expected 1987.

[Maddox-85]
The Ada+ code generator,
W. Maddox M. Barbacci, T. Newton, R. Stockton,
CMU Draft report, 1986.

[Meiling-83]
Portable Ada Programming System: back end compiler global design.
E. Meiling, J.S. Pedersen,
Document DDC 02/RPT/22, Dansk Datamatik Center, 1983.

[Moermans-85]
DAS direkt naar EM vertalen,
W.F. Moermans,
Unpublished Report, Department of Mathematics and Computer Science
Delft University of Technology, 1985.

[Murdie-83]
Functional specification of the York Ada Workbench compiler,
J. Murdie,
Technical report YCS 62
University of York, York, U.K., 1983

[Narfelt-84]
Towards a Kapse Database,
K.-H Narfelt, D. Schefstrom,
in: Proceedings IEEE conference on Ada applications and environments,
IEEE, 1984 pp 42 - 51.

[Narfelt-85]
Extending the scope of the program library,
K.-H. Narfelt, D. Schefstrom,

in: Proceedings of the Ada International Conference, 1985 Ada in Use, Cambridge University Press, 1985 pp 25 - 40.

[Nassi-80]
Efficient implementation of Ada tasks,
I. R. Nassi, A.N. Habermann,
Technical Report CMU-CS-80-103, Department of Computer Science Carnegie Mellon University, 1980.

[Nestor-86]
IDL, Interface Description Language Formal description,
J.R. Nestor, W.A. Wulf, D.A. Lamb,
Software Engineerings Institute
Carnegie-Mellon University Pittsburgh PA U.S.A, 1986.

[Newton-86]
An implementation of Ada generics,
T.D. Newton,
Technical Report CMU-CS-86-125
Department of Computer Science Carnegie Mellon University, 1986.

[Niet-84]
Een Nieuw Front End voor de DAS compiler,
M. de Niet,
Unpublished Masters Thesis (in Dutch), Department of Mathematics and Computer Science
Delft University of Technology, 1984.

[Nori-74]
The Pascal <P> Compiler: Implementation Notes,
K.V. Nori, U. Ammann, K. Jensen, H.H. Nageli,
Berichte des Instituts für Informatik
Eidgenössiche Technische Hochschule Zurich, 1984.

[Nowitz-82]
The Ada BreadBoard Compiler: Code generation,
D.A. Nowitz,
in: [BreadBoard-82].

[Pauw-83]
The first Code Generator Pass of the DAS compiler,
W.S. de Pauw,
Unpublished Masters Thesis, Department of Mathematics and Computer Science,
Delft University of Technology, 1983.

[Penello-80a]
A simplified Operator Identification Scheme for Ada,
T. Penello, F. DeRemer, R. Meyers,
Sigplan Notices 15 (7,8), July/August, 1980 pp 82 - 87.

[Penello-80b]
A simplification of 'A comment on "Operator Identification in Ada by Ganzinger and Ripken"',
T. Penello, F. DeRemer,
Sigplan Notices 16 (2) Februari, 1981 pp 2.

[Persch-80]
Overloading in Preliminary Ada,
G. Persch, G. Winterstein, M. Dausmann, S. Drossoupoulou,
in: Proceedings of the ACM-Sigplan Symposium on the Ada Programming Language
Sigplan Notices 15 (11) November, 1980 pp 47 - 56.

[Persch-81]
An LALR(1) Grammar for (Revised) Ada
G. Persch G. Winterstein, S. Drossopoulou, M. Dausmann,
Sigplan Notices 16 (3) March, 1981 pp 85 - 98.

[Persch-84]
A Portable Ada Tasking System for Single Processors,
G. Persch, H. S. Jansohn, R. Landwehr, J. Uhl, M. Dausmann,
GC ACM Programming Environments and Compilers,
Munich, April, 1984.

[Pierce-85]
Ada in the ECLIPSE Project Support Environment,
R.H. Pierce,
in: Proceedings of the Ada International Conference Paris, 1985 Ada in Use,
Cambridge University Press, 1985 pp 309 - 320.

[Quinn-82]
The Ada BreadBoard Compiler: Semantic Analysis,
M.E. Quinn,
in : [BreadBoard-82].

[Rain-84]
Avoiding trickle-down recompilation in the Mary-2 implementation,
M. Rain,
Software - Practice and Experience 14 (12) December, 1984 pp 1149 - 1157.

[Rasmussen-82]
Portable Ada Programming system: Separate Compilation Handler,
T. B. Rasmussen,
Dansk Datamatik Center, DDC 02/1982-10-18/a 1982.

[Rationale-79]
Rationale for the design of the Ada programming language,
Sigplan Notices, June, 1979 14 (6) part B.

[Reiss-83]
Generation of compiler symbol processing mechanisms from specifications,
Steven P. Reiss,
ACM Transactions on Programming Languages and Systems 5 (2) April, 1983 pp 127 - 163.

[Riccardi-85]
A Run-time Supervisor to Support Ada Tasking: Rendezvous and Delays,
G.A. Riccardi, T.P. Baker,
in: Proceedings of the Ada international Conference, 1985 Ada in Use,
Cambridge University Press, 1985 pp 329 - 342.

[Richards-80]
BCPL, the language and its compiler,
M. Richards, C. Whitby-Strevens,
Cambridge University Press, 1980.

[Roberts-81]
Task management in Ada: A critical evaluation for real-time multiprocessors,
E.S. Roberts, A. Evans, C.R. Morgan E.M. Clarke,
Software - Practice and Experience, 11 (10), October, 1981 pp 1019 - 1051.

[Röhrich-80]
Methods for the automatic construction of error correcting parsers,
J. Röhrich,
Acta Informatica 13 (2) February, 1980 pp 115 - 139.

[Rosen-83]
A kernel for tasks and rendezvous management in Ada,
J.P. Rosen,
Ada-Europe/AdaTec joint conference on Ada,
Brussels March, 1983.

[Rosen-86]
Une machine virtuelle pour Ada: le systeme d'exploitation,
J.P. Rosen,
These pour obtenir le grade de Docteur de l'ENST
Ecole Nationale Superieure des Telecommunications Paris 1986.

[Rosenberg-80]
The Charette Ada Compiler,
J. Rosenberg, D.A. Lamb, A. Hisgen and M. Sherman,
in: Proceedings of the ACM-Sigplan symposium on the Ada Programming Language
Sigplan Notices 15 (11) November, 1980 pp 91 - 100.

[Rosenblum-87]
An efficient communication kernel for distributed Ada run-time tasking supervisors,

D.S. Rosenblum,
Ada Letters 7 (2) March/April, 1987 pp 102 - 117.

[Roubine-82]
Lolita, a low level intermediate language for Ada,
O. Roubine, J. Teller, O. Maurel,
in: Proceedings of the AdaTec conference on Ada, Arlington October, 1982 pp 251 - 260.

[Rubine-82]
A Hybrid Ada Interpreter,
D.H. Rubine,
in: [BreadBoard-82].

[Schefstrom-86]
Integrating Ada in an existing environment; The arc example,
D. Schefstrom,
in: Proceedings of the Ada-Europe International Conference, 1986 Managing the Transition,
Cambridge University Press, 1986 pp 183 - 196.

[Schonberg-82]
An efficient method for handling operator overloading in Ada,
E. Schonberg, G.A. Fisher,
in: Proceedings of the AdaTec conference on Ada Arlington, October, 1982 pp 107 - 111.

[Sebesta-85]
Minimal Perfect Hash functions for reserved word lists,
R.W. Sebesta, M.A. Taylor,
Sigplan Notices 20 (12) December, 1985 pp 47 - 53.

[Sherman-80a]
A flexible semantic analyzer for Ada,
M. Sherman, M. Borkan,
in: Proceedings of the Sigplan Symposium on the Ada Programming Language
Sigplan Notices 15 (11) November, 1980.

[Sherman-80b]
An Ada code generator for VAX 11/780 with UNIX,
M. Sherman, A. Hisgen, D.A. Lamb, J. Rosenberg,
in: Proceedings of the ACM-Sigplan symposium on the Ada Programming language
Sigplan Notices 15 (11) November, 1980.

[SigAda-86]
Catalogue of Ada run-time implementation dependencies,
Prepared by Association for Computing Machinery, Special Interest Group on Ada,
Ada run-time Environment Working Group
November, 1986.

[Simpson-82]
The ALS Ada Compiler Front End Architecture,
R.T. Simpson,
in: Proceedings of the AdaTec conference on Ada Arlington, October, 1982 pp 98 - 106.

[Smith-84]
ANSI Standard Ada, quick reference sheet,
David A Smith,
Ada Letters 4 (1) July/August, 1984 pp 61 - 66.

[Someren-82a]
Computation of Addressing Information in the DAS code generator,
J. van Someren,
Unpublished Masters Thesis, Department of Mathematics and Computer Science
Delft University of Technology, 1982.

[Someren-82b]
Separate Compilation in DAS,
J. van Someren,
Unpublished Memorandum in: [Someren-82a].

[Stevenson-80]
Algorithms for Translating Ada Multitasking,
D. R. Stevenson,
in: Proceedings of the Sigplanb Symposium on the Ada Programming Language
Sigplan Notices 15 (11) November, 1980 pp 166 - 175.

[Stirling-85]
Follow set error recovery,
C. Stirling,
Software - Practice and Experience 15 (3) March, 1985 pp 239 - 257.

[Stockton-85]
Overload resolution in Ada+,
R.G. Stockton,
Technical Report CMU-CS-85-186
Department of Computer Science Carnegie Mellon University 1985.

[Stoneman-80]
Requirements for Ada Programming Support Environments, 'Stoneman',
U.S. Department of Defense 1980.

[Tai-80]
Comments on the suggested implementation of tasking facilities in the 'rationale for the design of the Ada programming language',
K.-C Tai, K. Garrard,
Sigplan Notices 15 (10) October, 1980 pp 76 - 84.

[Tanenbaum-83a]
Description of a machine architecture for use with block structured languages,
A.S. Tanenbaum, H. van Staveren, E.G. Keizer, J.W. Stevenson,
Informatica Report 81, Vrije Universiteit Amsterdam, 1983.

[Tanenbaum-83b]
A practical toolkit for making portable compilers,
A.S. Tanenbaum, H. van Staveren, E.G. Keizer, J.W. Stevenson,
Communications of the ACM 28 (9) September, 1983 pp 654 - 662.

[Teller-80]
Intermediate languages,
J. Teller,
Unpublished Manuscript Siemens November, 1980.

[Teller-81]
The architecture of a portable Ada compiler,
J. Teller, O. Roubine,
Unpublished Manuscript Siemens, 1981.

[Thall-82]
The KAPSE for the Ada Language System,
R.M. Thall,
in: Proceedings of the AdaTec conference on Ada Arlington, October, 1982 pp 31 - 47.

[Tichy-85]
RCS - a system for Version Control,
W.F. Tichy,
Software - Practice and Experience 15 (7), July, 1985.

[Toet-84b]
Code generation for expressions in the DAS compiler, some special topics,
W.J. Toetenel,
Unpublished Masters Thesis, Department of Mathematics and Computer Science
Delft University of Technology 1984.

[Uhl-82]
An attribute grammar for the semantic analysis of Ada,
J. Uhl, S. Drossoupoulou, G. Persch, G. Goos, M. Dausmann, G. Winterstein, W. Kirchgassner,
Springer Lecture Notes in Computer Science, LNC 139
Springer Verlag, 1982.

[Verdix-85]
VADS-operation manual, VADS version 5.1,
Verdix Corporation, 1985.

[Vonk-82]
A scanner/parser for DAS,
J.C. Vonk,
Unpublished Masters Thesis, Department of Mathematics and Computer Science
Delft University of Technology, 1982.

[Wallis-80]
Efficient implementation of the Ada overload rules,
P. Wallis. B. Silverman,
Information Processing Letters 10 (3) April, 1980 pp 120-123.

[Wand-82]
University of York Workbench Ada Compiler,
Status Report
I.C. Wand,
Ada Letters 2 (2) September/October, 1982 pp 99 - 104.

[Wand-87]
Facts and Figures About the York Ada Compiler,
I.C. Wand, J.R. Firth, C.H. Forsyth, L. Tsao, K.S. Walker,
Ada Letters 7 (4) July/August, 1987 pp 85 - 87.

[Weatherly-84]
A message based kernel to support Ada tasking,
R.M. Weatherly,
In: IEEE computer society, 1984 Conference on Ada Applications and Environments,
IEEE, 1984 pp 136 - 144.

[Wegner-68]
Programming languages, Information structures and Machine Organisation,
P. Wegner,
McGraw Hill, 1968

[Wellings-84]
Communication Between Ada Programs,
A.J. Wellings, G.M. Tomlinson, D. Keeffe, I.C. Wand,
In: IEEE Computer Society, 1984 Conference on Ada Applications and Environments, IEEE, 1984 pp 145 - 152.

[Welsh-86]
A model implementation of Standard Pascal,
J. Welsh, A. Hay,
Prentice-Hall International (Series in Computer Science) Englewood Cliffs 1986.

[Wetherell-81a]
LR - Automatic Parser Generator and LR (1) Parser,
C.S. Wetherell, A. Shannon,
IEEE Trans. Soft Eng, SE 7, 3 May, 1981.

[Wetherell-81b]
Problems with the Ada Reference Grammar,
C.S. Wetherell,
Sigplan Notices 16 (9), September, 1981 pp 90 - 104.

[Wetherell-82a]
The Ada BreadBoard Compiler: Lexical and Syntactical Analysis,
C.S. Wetherell,
In: [BreadBoard-82].

[Wetherell-82b]
The Ada BreadBoard Compiler: The pre-semantic pass,
M.E. Quinn, C.S. Wetherell,
In: [BreadBoard-82].

[Wichmann-79]
Ada is green,
B.A. Wichmann,
Computer Bulletin sept, 1979, pp 20 -21.

[Wilcox-85]
The interactive and incremental compilation of Ada using DIANA,
T.R. Wilcox, H.J. Larsen,
Unpublished Memorandum, Rational, 1985.

[Wilson-86]
The implications of Ada for configuration management and project support environment: Towards Adequate support,
G.M. Wilson, D.P. Youll,
in: Proceedings of the Ada-Europe International Conference, 1986 Managing the Transition,
Cambridge University Press, 1986 pp 91 - 104.

[Winterstein-85]
A portable optimising compiler for a production quality European Apse
Initial study,
Dr. Winterstein - Systeam, Karlsruhe W. Germany
Systems Designers Scientific, Surrey G.B. 1985.

[Wirth-77]
Modula - a language for modular multiprogramming,
N. Wirth,
Software - Practice and Experience 7, 1977.

[Wolverton-84]
A perfect hash function for Ada reserved words,
D. A. Wolverton,
Ada Letters 4 (1) July/August, 1984 pp 40 - 44.

[Zang]
A proposal for implementing the concurrent mechanisms of Ada,
X. Zang,
Sigplan Notices 21 (8) August, 1986 pp 71 - 79.

Curriculum Vitae

J van Katwijk

Geboren 24 februari 1945, in Amsterdam. Bezocht het Hervormd Lyceum te Amsterdam en deed eindexamen HBS-B in 1964.
Vervulde in de jaren 1965/1966 de militaire dienstpicht.
Studeerde aan de Technische Hogeschool te Delft. Electrotechniek in 1966/1967, Wiskunde 1967 - 1971. Studeerde af in 1971.
Was na het afstuderen werkzaam als wetenschappelijk medewerker bij de Onderafdeling der Wiskunde van de Technische Hogeschool Delft.
Sinds 1 december 1985 Universitair Hoofddocent bij de vakgroep Informatica van de Faculteit Wiskunde en Informatica van de Technische Universiteit Delft op het vakgebied Programmeertalen, vertalerbouw en systeemprogrammatuur.