LOLITA - A Low Level Intermediate Language for Ada+*

Olivier Roubine, Cii-Honeywell Bull
Joachim Teller, Siemens A.G.
Olivier Maurel, Alsys S.A

1 Introduction

Under the auspices of the Commission of the European Communities, the development of a portable root for Ada compilers has been undertaken by Alsys S.A, Cii-Honeywell Bull and Siemens A.G.

One of the most important aspects of such an effort is the interface between the (essentially machine-independent) root of the compiler and the code generators. If portability is to become a reality, the writing of a code generator should be an easy task, and the internal program representation must facilitate it. This requirement may be at odds with the desire to provide an interface between Ada compilers and sophisticated tools in a programming support environment.

Consequently an Ada compiler has to provide two levels of intermediate languages. It is generally accepted that an internal program representation derived from the abstract syntax has the appropriate level and structure to represent the high level intermediate language. No common approach for a corresponding low level intermediate language has been encouraged, attempted, or achieved so far. Some implementors have been trying to stretch the high level intermediate language in order to fit the needs for a lower level representation. Others have taken the opposite approach, i.e., they have been designing high level architectures in order to obtain an intermediate language in terms of the instruction set for an Ada machine.

The approach proposed in this paper aims at an intermediate language that is appropriate to serve as an interface at the portability level. It is not directly derived from the corresponding high level intermediate language, but is the result of a selfstanding design.

+This work was partially supported by the Commission of the European Communities.

*Ada is a trademark of the United States Department of Defense (Ada Joint Project Office).

2 Intermediate Languages

One of the major issues in the design of portable compilers is the notion of intermediate languages. They are used for the internal representation of a program, and serve as an interface between the front-end and the back-end.

This assumes that it is easily possible to divide the compilation process into a source language oriented (and hence machine independent) analysis part and a target machine oriented (and hence machine dependent) code generation part. Consequently it is deduced that the task of transporting a compiler just means that the machine-dependent back-end has to be rewritten for a particular target machine. In order to be able to reap the maximum benefit from such an approach, it is also required that rewriting of the code generator be as straightforward and cheap as possible. This requirement has to be taken into account in the design of the intermediate language for a portable compiler. Because of the possible economical benefits of portable compilers, major emphasis has been put into the design of intermediate languages.

2.1 Design Approaches

Basically there exist two particular approaches for the design of an intermediate language.

- A top-down approach, which defines the intermediate language in terms of the source language
- a bottom-up approach, which defines the intermediate language in terms of a hypothetical target machine.

While the first approach is more related to compiler technology and nearly a by-product of the design of a modular compiler, the second approach is more related to abstract machine modelling. In the latter case the intermediate language is equivalent to the machine language for an abstract machine. In order to avoid any confusion of terms which might lead to misunderstandings, intermediate languages resulting from the application of the first approach will be called internal program representation (IPR), while those resulting from the second approach will be called abstract machine languages (AML). The two design approaches, although starting from opposite directions, may possibly yield equivalent results. On the other side the differences between the results of the two approaches may be considerable, if regarding an IPR representing a "syntax tree" and an AML for a one-address machine.

2.2 Technical Approaches

In the following some of the best known and most important approaches are outlined.

The earliest works on intermediate languages dealt more or less only with the internal representation of expressions in the form of prefix or postfix notations.

The next step was quadruples containing explicitly named results and applicable to language constructs other than expressions. Another technique introduced a parameterization facility by regarding each intermediate language statement as an n-tuple representing a macro call. In this case the different code generation algorithms were described in the bodies of the corresponding macros. While the approaches described above were not languages in the usual sense, the ones descibed below qualify as such.

The first really comprehensive and still most famous approach to the specification of an intermediate language was UNCOL [Ste 60]. It was based on economical considerations and tried to reduce the cost for implementing n languages on m machines from n x m to n + m. UNCOL was meant to become a unique intermediate language for any source language and target machine.

Although this approach never actually succeeded (even no definite specification of UNCOL was ever produced), the basic idea has never been abandoned and is being revived from time to time. A recent example is to a certain degree TCOL [New 79].

While it seems that the design of universal intermediate languages is at the least problematic, some other more specific approaches have been very successful. Most of them belong to the class of abstract machine languages, some to internal program representations.

- OCODE is an intermediate language which has been designed to implement BCPL [Ric 71]. Describing a one-address machine, its simplicity contributed to a wide availability of BCPL. The rewriting of the code generator was an extremely simple task and required an effort of a few man-months.

- PCODE [Ber 78] is an intermediate language that has been designed to implement Pascal. PCODE is describing a simple stack machine, which may also be implemented with a reasonably small effort on a new machine.

- JANUS [HW 78] is an intermediate language that has not been designed for a particular language. It describes an elaborate stack machine.

- DIANA [GW 81] is an intermediate language that has been designed to implement Ada. Its design is not based upon a particular machine model but derived from the abstract syntax. Hence it is tree-structured and decorated with particular attributes.

2.3 Evaluation

The OCODE approach suffers from the fact that the underlying machine is at much too low a level. Mapping of a one-address machine onto a more elaborate machine leads to unacceptable inefficiencies.

The PCODE approach is based on a machine model that is too rigid and also does not allow for the generation of high quality code, as some common optimizations such as the evaluation of common subexpressions are made overly difficult.

The JANUS approach not being attached to a particular source language suffered from the fact that its underlying machine model had only limited properties with respect to data types and other language features, e.g., procedure calls. For the implementation of a new language with new properties it has to be redesigned or at least enhanced.

The DIANA approach which provides a totally machine independent representation of Ada programs has the drawback that the task to generate code from DIANA is a tremendous job. It is at too high a level to be seriously regarded as a portability interface for compiler construction.

The approach presented in this paper is an attempt to build on those presented above, retaining their main advantages and palliating their deficiencies. It belongs to the class of internal program representations being designed for a particular source language; it is at a much lower level than DIANA and is not related to any abstract machine model.

3 Background

3.1 Requirements of the European Ada Compiler Project

The task of designing an intermediate language for an Ada compiler is far from being trivial, as several factors come into play. We have to consider several aspects and weigh them against the specific goals of the compiler.

The European Ada Compiler has two major objectives:

- provide a portable root that can be retargetted at little cost
- generate production quality code that can compete with the best compilers available.

The first goal introduces the need for an intermediate language that is not dependent on the features of a particular machine (portability of the root), while at the same time allowing the code generator to be fairly simple (ease of retargetting).

The need for an optimizing compiler quickly led us to the conclusion that optimizations would have to be performed at several different levels, as some analysis is more easily performed when reasoning in terms close to the source language itself (an example being the elimination of unnecessary checks) while some optimizations yield better results when applied to a lower level representation of the program, where in particular address calculations have been made explicit (so that they can be considered as candidates for common sub-expression elimination).

As a result, the compiler hinges on two intermediate representations of the program, expressed in a high level Abstract Intermediate Language, and a low level intermediate language (LOLITA), which is the subject of this paper.

This dichotomy had a fortunate consequence: since there exists a higher level representation of the program, we felt a great freedom in pushing the low level language towards a lower level, as long as machine independence permitted it. If this level became too low for certain optimizations, it was a good indication that such optimizations should be performed on the high level intermediate language.

3.2 Compiler Architecture

Our compiler is composed of three major parts: the Analyzer, Expander and a Back-End.

The Analyzer performs lexical, syntactic and semantic analysis, identification and type implementation.

The Expander performs data allocation and the transformation of the high into the low level intermediate language. In addition two optional optimization steps may be performed.

These two phases constitute the so-called Root compiler. This Root represents the portable part of the compiler; however, it is not absolutely machine-independent but parameterized in terms of implementation choices. This is in particular true for type implementation and data allocation. While type implementation has to reflect actual machine properties, data allocation has to implement the choices made for the runtime layout of data.

The Back-End which represents the traditional machine-dependent part is performing the mapping of the low level intermediate language onto the object code of the corresponding target machine. Usually this comprises tasks such as register allocation, instruction sequence selection, data addressing, object editing, and some peephole optimization.

The major interfaces between these three parts are built by the two levels of intermediate languages as well as corresponding symbol tables, one for each level.

4 Design Philosophy

The two major goals for the compiler, ease of retargetting, and high quality code must be kept in mind when examining the design of the low level language.

More specifically, the compiler root must be easily adaptable to any machine within a large class of "conventional" architectures, typically machines with directly addressable memory and a set of registers.

4.1 Flexibility

Rather than trying to conceive an abstract machine model, whose machine code would be the intermediate language of the compiler, a less rigid approach was preferred, whereby the intermediate language would consist of a set of features that could be combined in several ways to express the execution semantics of a given program. In other terms, there is no single representation of a given Ada program in LOLITA; rather, it is possible to custom-tailor the intermediate representation for a specific implementation, while keeping an essentially machine independent language. This will be best examplified by a most classical aspect, the access to non-local subprogram frames: certain implementations may decide to use static links while others may use a display kept either in memory or in registers. The particular choice depends on a variety of criteria: number of registers available, addressing modes of the hardware, uniformity of the register set, etc... Three alternatives could be considered for an intermediate language:

(a) The access path to a particular frame is kept implicit: for instance, a variable is indicated by a pair (level, displacement). This corresponds to a higher level IL, where access paths cannot be considered as common subexpressions.

(b) A particular execution model is chosen, and the IL reflects this choice. Such a rigid approach corresponds to an "abstract machine" definition and is considered overly preemptive.

(c) Features are provided to express offset and indirection. With these features, it is possible to express any access path explicitly, although access paths may turn out differently on different implementations. This corresponds to the approach taken in LOLITA.

4.2 Structure

An important aspect of an intermediate language is its structure: most conventional intermediate languages intended for code generation consist of a linear sequence of n-tuples (generally triples or quadruples). LOLITA on the contrary is tree-structured (hence its name, Low Level Intermediate Tree for Ada): an operation is typically described by an operator node with one or more sons reflecting the operands. This choice reflects our concern for machine independence, ease of code generation and powerful optimizations: a linear sequence forces a specific representation of control structures, in a manner that often assumes certain properties of the machine: it later becomes awkward to generate code for a different machine. A good example is the control of loops: in a linear intermediate code, a simple goto will be the only indication that repetition is to occur. The code generator has to strain itself to an undesirable extent in order to generate efficient instructions such as "branch on count".

There are several places where it is more convenient to express the flow of control structurally. This is especially the case for the tasking features of Ada, e.g. the select statement, where the flow of control is determined by the occurrence of some events external to the task. A tree structure permits a more flexible processing, for example by allowing decisions to be made on the basis of all the immediate descendants, or to be deferred until all the subtrees have been visited.

Lastly, the IL structure must facilitate the work of the optimizer. A tree structure permits code transformation, or identification of specific patterns to be performed more easily. In particular, the structure of the intermediate language must be compatible with state of the art code generation techniques, such as those developed by Glanville [Gla 78] and Ganapathi [Gan 80]. Such methods are directly applicable when the LOLITA tree is traversed in preorder.

4.3 Level of the Language

One has to strike a balance between machine independence, which drives the intermediate language towards a higher level, and ease of code generation which drives it to a lower level.

The main idea was to make the largest number of operations appear explicitly, while retaining only a small number of primitive operators: when a language construct was examined, we first tried to express its semantics in terms of the primitives already defined. Only if this turned out to be inconvenient or awkward did we try to introduce a new feature in the language.

It is difficult to illustrate this philosophy other than by discussing the rationale for certain features. As this is precisely what we intend to do in the next section, we refer the reader to the various details presented below.

5 The Salient Features of LOLITA

Our purpose is not to present here a complete overview of LOLITA, but rather to discuss what we consider to be some of its interesting features, especially as far as their design is concerned.

These features are classified in four broad categories: program structure, operands, expressions, and statements.

(Notation: in the following, particular LOLITA constructs are presented by lower-case identifiers. Upper-case identifiers are used to represent sorts, that can stand for various subtrees).

5.1 Program Structure

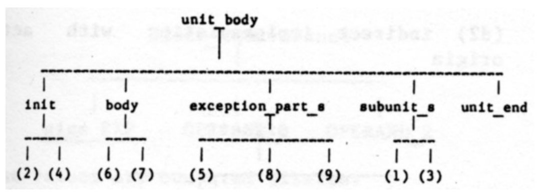
The structure of a source program does not always reflect the execution order, as the body of subprograms may be interspersed with other declarations, or exception handlers may appear at the end of a block.

In order to suppress unnecessary inefficiencies (jumps around pieces of code that are not executed in-line), the structure of a program represented by a LOLITA tree does not always reflect trivially that of a source program: for a given subprogram, the exception handlers, as well as those of any nested package body or block are grouped together after the body itself. Similarly, the bodies of any nested subprograms appear after the handlers. Thus, the representation of the following program fragment

```
procedure P is
   procedure Q is (1) end;
   package R is (2) end;
   package body R is
      procedure S is (3) end;
   begin
      (4)
   exception
      (5)
   end;
begin
```

```
        (6)
      begin
        (7)
      exception
        (8)
      end;
    exception
      (9)
    end;
```

would be



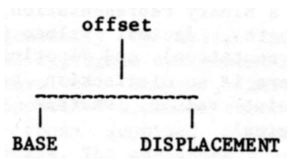
unit_end contains the so-called cleanup actions for each nested block, and for the subprogram. These are statements to be executed when control leaves a unit other than by its normal end. They are typically storage management operations.

5.2 Representation of Operands

The term operand is used to designate objects that have a runtime representation. Such objects may be objects in the sense of Ada, as well as objects generated by the compiler. At the high level, operands are merely references to an object that has been declared and are mostly represented by means of references to corresponding symbol table entries. At the low level, operands are described in terms of address calculations designating a memory location. These address calculations are called access paths.

Access paths designate objects which may be program units themselves, be located in program units belonging to the same or to a different compilation unit, or may be components of compound data structures. In fact no difference is made whether an access path is derived from an equivalent Ada construct or a compiler-made implementation choice.

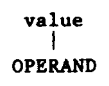
The general notion of access paths is based upon the following basic construct:



The BASE is itself an address computation, while the DISPLACEMENT indicates a value to be compounded with the base. It may correspond to an offset in storage units or in bits.

Different incarnations of this construct exist in the low level intermediate language reflecting the particular context where they occur.

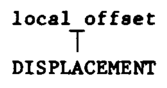
The access to values being denoted by an access path is obtained by means of an explicit dereferencing operator:



In the following some simple examples are given to illustrate the way access paths can be expressed in LOLITA.

(a) access to local data

Data Allocation assigns objects to different storage classes, one of them being the class (stack) frame. For the access to data belonging to the current frame no explicit base is specified. The reference merely specifies the displacement.

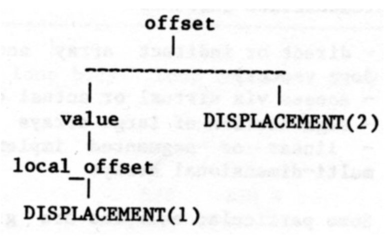


Before final addressing, DISPLACEMENT corresponds to a reference to a corresponding entry in the symbol table. Thereafter it will be replaced by a value representing the displacement in terms of storage units.

(b) access to enclosing scopes

The access to data being allocated in another frame, i.e., belonging to a statically enclosing unit may be implemented by means of a static link or a display vector. Depending on which implementation is chosen, it is possible to specify an appropriate access path.

An example for a static link implementation is given below:

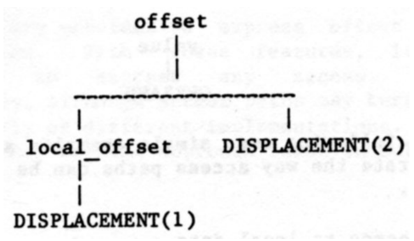


(1) offset of the static link in the current frame
(2) displacement of the object in its containing frame (here 1 level above).

It is obvious how an access path would look like if several static links had to be employed in sequence.

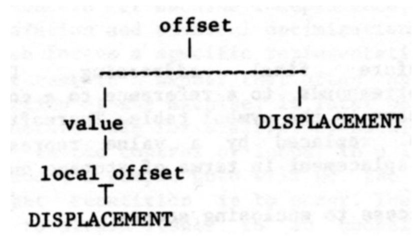
(c) access to record components

The access to a component of a record being allocated in the current stack frame is represented as follows:



(1) offset of the base of the record in the current frame
(2) displacement of the field within the record.

Assuming that the record object is designated by an access variable being allocated in the current frame yields the following representation:



Comparing the above access path with that specified in (b) also illustrates the statement made at the beginning about equivalent access paths being generated for different source-level constructs.

(d) access to array elements

The access path generated to denote the element of an array depends on several implementation choices:

- direct or indirect array access (e.g., dope vector)
- access via virtual or actual origin
- segmentation of large arrays
- linear or segmented implementation of multi-dimensional arrays

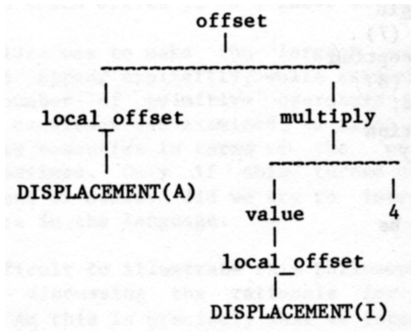
Some particular examples are given below:

Considering the array A of type

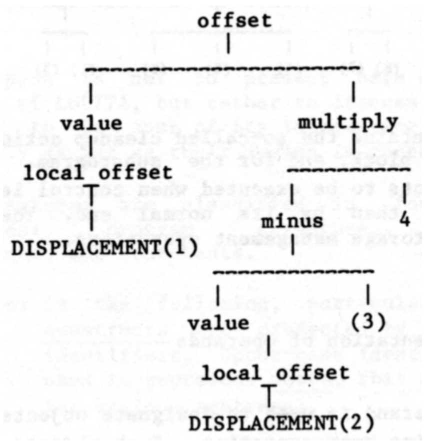
array (INTEGER **range** <>) of INTEGER

access to the element A(I) is shown in 2 cases:

(dl) direct implementation with virtual origin



(d2) indirect implementation with actual origin



(1) offset of the dope for A
(2) offset of the variable I in the current frame
(3) value of the lower bound of the array

(checks that might be needed, e.g., index checks are described in 5.6).

5.3 Expressions

The representation of expressions in LOLITA is rather typical of a tree-structured language, at least as far as "simple" values are involved.

There are three categories of simple values, corresponding to the hardware types most frequently encountered: these are binary values (addresses, numbers with a binary representation, bitstrings of a fixed length), decimal values (numbers with a decimal representation), and floating point values. Note that there is no distinction between integer and fixed-point values, whether represented in binary or decimal.

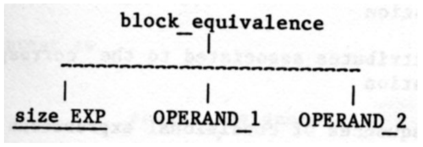
Since the way code can be generated for various expressions is widely different for the different types of values, LOLITA provides different operators for each type (rather than identifying the type by an attribute). This is particularly useful to identify certain patterns, e.g.,

```
x := x + 1
```

can be treated specially for binary operands, but not for decimal or floating point ones.

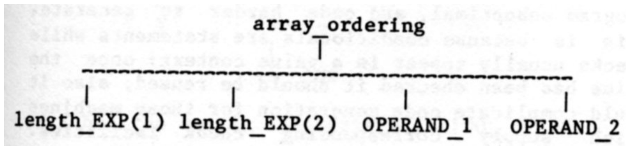
Another area of interest is the treatment of compound values, especially arrays. The two cases in point are the equivalence and ordering operations.

LOLITA contains a "block-equivalence" construct which expresses the comparison of two blocks of data over a certain size that can be expressed dynamically.



The values are compared bitwise.

Array ordering, on the other hand, has a very specific semantic, since the lengths of the operands may determine the result of the expression. A special operator is therefore adequate:



Note that compound values are always designated by their addresses, an additional necessity for specialized constructs.

5.4 Statements

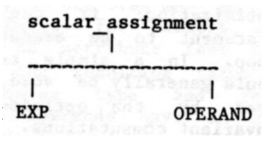
At the level of the low level intermediate language, statements have to be divided into assignment, simple and compound statements. The representation of simple statements is quite straightforward and requires no particular presentation. However of particular interest is the representation of assignments and the structure of compound statements.

5.4.1 Assignment

Assignments may generally be split into scalar, array, record and aggregate assignments, according to the operands involved. However, there exists no one-to-one correspondence between these types of assignments and those supplied in the low level intermediate language. The assignments provided at this level have a layout which is more oriented to reflect the particular implementation properties of an assignment.

(a) scalar assignment

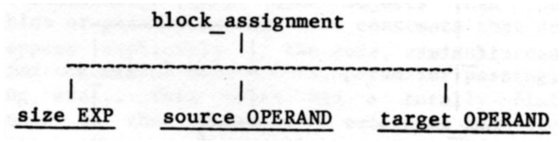
Except for the fact that the order of the operands is reversed to reflect the execution order more closely, there is little special about this construct, which looks as follows:



The scalar assignment can also act as an expression (representing the value that has just been assigned).

(b) Assignments of Compound Objects

A special construct is used to express the assignment of storage.

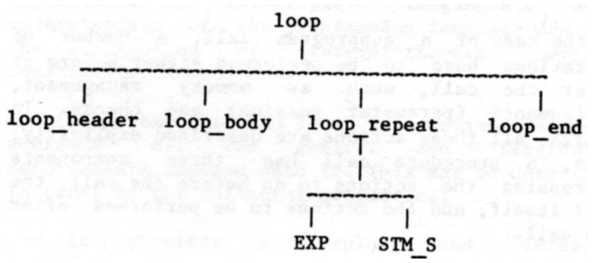


This construct can be used in particular for array assignments. However, the possibility of overlapping slices must be catered for: an attribute indicates whether an assignment by increasing or decreasing addresses is called for, or whether a test must first be generated (because there may be an overlap that cannot be determined at compile-time).

For more complex representation of objects, a combination of these and other statements (e.g. loops) can be used.

5.4.2 Loops

All loops appearing in a program, and most of those generated by the compiler itself are represented using a single construct



where:

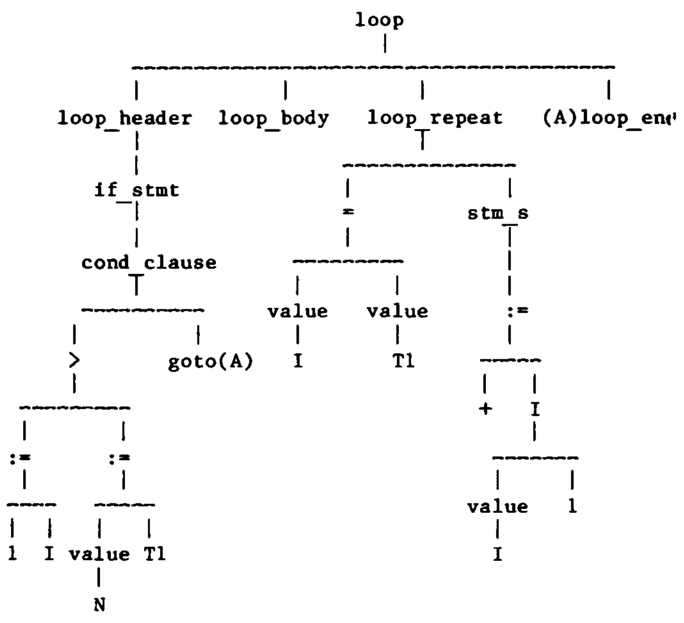
loop_header	is a list of statements representing some initialization, to be executed once, before the loop.
loop_body	is the list of statements that has to be executed at each iteration.
loop_repeat	contains a condition, to be evaluated after each iteration: a true value causes exit from the loop. It may also

contain a list of statements to be executed before the next iteration (typically increments).

loop_end is typical of our preoccupation for optimization: it can contain any statement to be executed after the loop. In a simple translation, it would generally be void, but can be used by the optimizer for moving invariant computations.

Example:

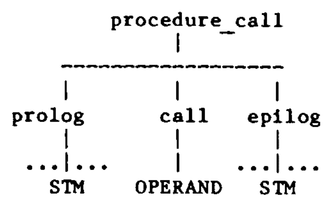
for I **in** 1..N **loop** ... **end loop**;



(Note: the notation used here takes some liberty with the standard formalism, for the sake of brevity and readability).

5.5 Subprogram Calls

In the case of a subprogram call, a number of operations have to be performed either before or after the call, such as memory management, assignments (parameter passing) and checks. In LOLITA, all these actions are described explicitly; thus, a procedure call has three components expressing the actions to do before the call, the call itself, and the actions to be performed after the call.



A function call has a similar structure, but also represents the operand containing the result after the call.

5.6 Checks

The kind of checks required by Ada, which may result in actions at runtime are in particular constraint and numeric checks. While the treatment of numeric checks is largely machine-dependent, constraint checks may be treated in a basically machine-independent way. Hence corresponding constructs are provided in LOLITA.

There exist in general three options for their representation

- as attributes associated to the corresponding operation
- as sequences of conditional expressions
- as separate operators

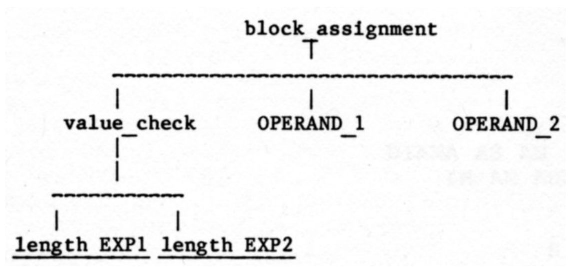
The first approach is in contradiction with the design principles of the low level intermediate language, to make the semantics of an operation as explicit as possible. The second approach has been abandoned as it makes the representation of the program suboptimal, and code harder to generate. This is because conditionals are statements while checks usually appear in a value context: once the value has been checked it should be reused; also it would complicate code generation for those machines which supply corresponding check facilities. Therefore it has been decided to represent checks as explicit operators. However, instead of defining a check operator for each of the checks described for Ada, their particular semantics have been taken into account.

LOLITA provides the following check operators:

- value checks
- range checks
- discriminant checks
- access checks

The general semantics of the above checks are to check a value against a given constraint. If the constraint is violated a corresponding exception is raised. The successfully checked value is the result of the check (except in the case of a discriminant check, where the value is the base of the record).

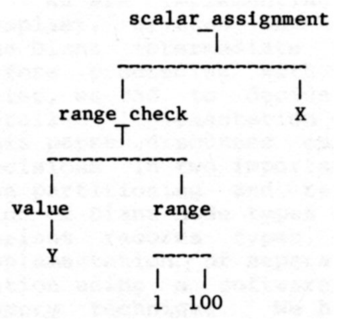
A value check expresses a simple comparison. For example, to check that two arrays have the same length in an assignment, we would use



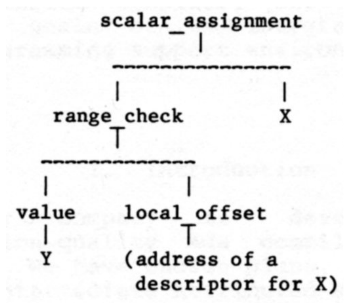
A range check tests that a scalar value is within a given interval, expressed either as a pair of values, or as the address of a runtime description. Thus,

X := Y;

could appear as



if X had the static constraints 1..100, or as



if X had a dynamic constraint.

(Note: although we did not express this in the above examples, the range and value check operators are typed as is the case for the binary operators).

Although a test for a null division can be expressed with a value check, the case of a numeric overflow is treated differently: we assume some help from the hardware, and any expression has a boolean indicator, set whenever the code generator must take actions so that the result of the expression be checked. Of course, if the hardware actually traps, the code generator does not have to insert extra instructions (e.g., to test a condition code).

5.7 Complements

The tree structure itself does not always contain sufficient information to allow for the easy generation of efficient code. It is complemented in two ways.

First, each node of the tree may be augmented by specific attributes: for example, all expression constructs have an attribute that describes the properties of the resulting value in terms of a particular machine type and representation. Similarly, all operands have an attribute that indicates whether the address they represent is expressed in bits or in storage units. Because most of the information is in fact represented in the tree structure itself, only few attributes are needed on each node.

The second informational structure that is added to the LOLITA tree is a low level symbol table. This table consists of a set of entries that describe the runtime properties of all the objects accessed by a compilation unit. Such objects can be variables or parameters, but also constants that do not appear explicitly in the code, various areas used for constraint descriptors, parameter passing, tasking, etc... This table has a totally flat structure in that there is no reference from one entry to another. Each entry is referenced when needed by an attribute in the LOLITA tree. The information contained in the table depends on the kind of object it corresponds to, but is generally concerned with addressing, indicating e.g., an offset, a register name or an external name to be resolved later. Such a table has the additional advantage of allowing the final resolution of addresses to be deferred until later stages of the compilation.

6 Conclusion

The design of a low level intermediate language for a portable, production quality Ada compiler has been described. Although this intermediate language has been developed in the framework of a particular implementation project, its design is claimed to be of general validity. The design has been based upon a vast experience in the development and application of intermediate languages,a deep understanding of the programming language Ada and the application of state-of-the-art compiler construction techniques.

We think that this low level intermediate language achieves all the general and particular requirements imposed onto it. This may be described in terms of the following properties:

- it provides a complete and detailed description of the semantics of an Ada program;
- it provides a machine independent representation of a given Ada program;
- it provides a flexible notation in particular with respect to different implementation strategies which may be chosen;

- although the implementation is still an ongoing effort, it is sufficiently advanced to give us the confidence that the code generation effort is supported rather than hindered by the language.

Acknowledgments

Many of the people involved in the design of the European Ada Compiler had a significant influence on LOLITA, especially: A. Buechler, D. Couprie, G. Diessl, B. Dion, M. Gart, J-C. Heliard, M. Pellissier, H. Raffler and R. Tillo-Schneider. In addition, the advice of B. Brosgol, G. Fisher, P. Lahtinen, R. Landwehr, E. Meyling and G. Romanski has been particularly appreciated. Lastly, this paper owes a lot to the skillful typing of Jocelyn Mahdjate.

* * * * *

REFERENCES

[Ber 78] Berry, R.E.: "Experience with the Pascal P-compiler", Software - Practice and Experience, Vol. 8, No. 5, September-October 1978.

[GW 81] Goos, G. and Wulf, W.A., (ed): "DIANA Reference Manual", Technical Report CMU-CS-81-101, Carnegie-Mellon University, March 1981.

[Gan 80] Ganapathi, M.: "Retargetable Code Generation and Optimization using Attribute Grammars", Technical Report 406, Computer Sciences Department, University of Wisconsin, Madison, 1980.

[Gla 78] Glanville, R.S.: "A Machine Independent Algorithm for Code Generation and its use in Retargetable Compilers," Technical Report UCB-CS-78-01 University of California, Berkeley, 1978.

[HW 78] Haddon, B.K., and Waite, W.M.: "Experience with the Universal Intermediate Language Janus", Software - Practice and Experience, Vol. 8, No. 5, September-October 1978.

[New 79] Newcomer, J.M. et al.: "TCOL-ADA: Revised Report", Technical Report CMU-CS-79-128, Carnegie-Mellon University, June 1979.

[Ric 71] Richards, M.: "The Portability of the BCPL Compiler", Software - Practice and Experience, Vol. 1, No. 2, April-June 1971.

[Ste 60] Steel, T.B.: "UNCOL, Universal Computer Oriented Language revisited," Datamation Vol 6, Jan- Feb 1960.