Diss. ETH Nr. 10277

A Programming Language for Vector Computers

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH
(ETH Zürich)

for the degree of
Doctor of Technical Sciences

presented by
Robert Griesemer, Dipl. Informatik-Ing. ETH
born June 9, 1964
citizen of Güttingen, Thurgau

accepted on the recommendation of
Prof. Dr. H. Mössenböck, examiner
Prof. Dr. N. Wirth, co-examiner

1993

Acknowledgements

This work would not have been possible without the help and encouragement of many people. I am most indebted to Prof. H. Mössenböck for his continuous support and his insistence on clarity and simplicity. If this thesis is more understandable now, this is due to his constructive criticism. I also would like to thank Prof. N. Wirth for a liberal supervision of this project and for posing the right questions at the right time. His comments lead to a much better and clearer understanding of what programming is all about.

It is a pleasure to mention my colleagues of the Institute for Computer Systems; they cared for an agreeable and inspiring environment and many of them contributed to this work. Clemens Szyperski and Josef Templ constructed the tools for preparing this document, namely the excellent Write text processing system (Clemens) as well as the unique graphics editor Kepler (Josef). Marc Brandis was a well-versed discussion partner in all kinds of compiler related topics. The Oberon-2 compiler used for the implementation was developed by Régis Crelier and would pass industrial strength tests. Matthias Hausner readily adapted the Write editor to even higher requirements. Ralph Sommerer implemented some tools to create amazing Mandelbrot pictures from incredibly long Cray output. Wolfgang Weck and Phillipp Heuberger helped with their experience in theoretical issues.

Bernd Mösli and Urs von Matt gave valuable hints on the requirements of numerical applications. Marc Brandis, Régis Crelier, Kathrin Glavitsch, Josef Templ, and my brother Marcel carefully proofread parts of this thesis. Albert Weiss and Immo Noack cared for a frictionless operation of the hardware. For various fruitful discussions and suggestions I would also like to thank Dr. S. E. Knudson, B. Loepfe, Prof. Dr. B. Sanders, and Dr. A. Schei. Several students contributed also to this work, namely Jürg Bolliger, Hugo Hack, Patrick Spiess, and Roger Waldner.

Last but not least, my warmest thanks go to my life companion Sabine for her patience and love through all the time.

Contents

Abstract

This thesis introduces and elaborates on specific language constructs that allow a simple programming of vector computers and help to gain a better understanding for these programs. Thereby the emphasis lies on the support of explicitly vectorizable statements as well as on a concept for parameter passing adapted to the needs of numerical applications.

Vector computers provide powerful instructions for the processing of whole vectors. The speed of programs is often increasing by orders of magnitude if these programs allow the use of such instructions, i.e. if they are vectorizable. In order to make a program run faster, a compiler usually tries to vectorize its innermost loops. Unfortunately, the dependence analysis required therefore is quite complicated and often cannot be performed completely. The thesis therefore proposes a simple language construct allowing the explicit specification of independence and thus the parallel execution of statements. Hence, this language construct is much easier to vectorize than loops. It improves the readability and security of programs without reducing the quality of the generated code.

The main application area of vector computers are numerical applications of linear algebra. A problem arising with those programs is that parts of matrices such as rows, columns or diagonals must be passed as arguments to a subroutine. Yet, most programming languages do not support such a flexible way of parameter passing. Array constructors offer a simple and safe way to solve this problem.

The second part of the thesis focuses on the description of an experimental programming language called Oberon-V and of an appropriate cross-compiler for the Cray Y-MP. Oberon-V includes a subset of the language Oberon, which has been extended by the language constructs mentioned above. Compared to traditional compilers for vector computers, the Oberon-V compiler excels by its compactness and efficiency. Detail problems of implementation were solved in a new and more simple way: some of the achievements were a new way of generating symbol files to support separate compilation, the optimization of the generated code by eliminating redundant computations (common subexpression elimination) and the reorganization of instructions to increase the execution rate (instruction scheduling). The thesis finally investigates and judges the code quality of Oberon-V programs in comparison with corresponding Fortran programs.

Kurzfassung

In dieser Arbeit werden spezielle Sprachkonstrukte eingeführt, die das einfache Programmieren von Vektorrechnern erlauben und zu einem besseren Verständnis dieser Programme beitragen sollen. Im Vordergrund steht dabei die Unterstützung von explizit vektorisierbaren Anweisungen sowie ein an die Bedürfnisse numerischer Applikationen angepasstes Parameterübergabe-Konzept.

Vektorrechner stellen leistungsfähige Instruktionen für das Bearbeiten ganzer Vektoren zur Verfügung. Programme werden oft um Grössenordnungen schneller, wenn sie von solchen Instruktionen Gebrauch machen können; d.h. wenn sie vektorisierbar sind. Zu diesem Zweck versucht ein Compiler gewöhnlich die innersten Schleifen eines Programms zu vektorisieren. Die dazu notwendigen Abhängigkeits-Analysen sind kompliziert und können häufig auch nur unvollständig ausgeführt werden. Es wird ein einfaches Sprachkonstrukt vorgeschlagen, welches die Unabhängigkeit und somit die parallele Ausführbarkeit von Anweisungen explizit auszudrücken erlaubt, und deshalb wesentlich einfacher vektorisierbar ist als Schleifen. Es wird gezeigt, dass damit die Qualität der Programme bezüglich Lesbarkeit und Sicherheit verbessert werden kann ohne dabei die Qualität des erzeugten Codes zu vermindern.

Hauptanwendungsgebiet von Vektorrechnern sind numerische Applikationen aus dem Bereich der linearen Algebra. In solchen Programmen stellt sich oft das Problem, dass Teile von Matrizen, z.B. Zeilen, Spalten oder Diagonalen als Argumente einem Unterprogramm übergeben werden müssen. Die meisten Programmiersprachen unterstützen eine solch flexible Art der Parameterübergabe nicht. Mit Hilfe von Array-Konstruktoren lässt sich dieses Problem einfach und vor allem sicher lösen.

In einem zweiten Teil der Arbeit wird eine experimentelle Programmiersprache (Oberon-V) sowie ein dazugehörender Cross-Compiler für die Cray Y-MP vorgestellt. Oberon-V umfasst eine Teilmenge der Sprache Oberon, welche um die erwähnten Sprachkonstrukte erweitert worden ist. Der Oberon-V Compiler besticht durch seine Kompaktheit und Effizienz im Vergleich zu herkömmlichen Übersetzern für Vektorrechner. In der Implementierung wurden diverse Detailprobleme auf zum Teil neue und einfachere Art und Weise realisiert. Dazu gehören eine neue Art der Erzeugung von Symbol-Files für die Unterstützung getrennter Übersetzung, die Optimierung des erzeugten Codes durch Entfernen redundanter Berechnungen (Common Subexpression Elimination) sowie das Umordnen von Instruktionen zur Steigerung der Ausführungsgeschwindigkeit (Instruction Scheduling). Schliesslich werden Oberon-V Programme sowie die erreichte Codequalität im Vergleich zu entsprechenden Fortran-Programmen untersucht und kritisch beurteilt.

1 Introduction

"As soon as an Analytical Engine exists, it will necessarily guide the future course of science. Whenever any result is sought by its aid, the question will then arise – by what course of calculation can these results be arrived at by the machine in the shortest time?"

Charles Babbage – The Life of a Philosopher, 1864

With the development of vector computers and massively parallel machines, highly computing-intensive applications have become feasible within reasonable time bounds. Since numerical programs constitute the major part of all applications on these machines, the programming language Fortran is used in most cases, despite its shortcomings.

While it does not seem clear yet which is the best way to program massively parallel machines, the programming of vector computers is comparatively well understood. This thesis concentrates on vector computers only. Since Fortran 77 [Brainerd et al. 1978] provides no special language support for these machines, an optimizing compiler typically tries to *vectorize* innermost DO loops; i.e. it tries to restructure the program in order to allow vector instructions to be used instead of scalar instructions. It is highly desirable to have as many vectorizable loops as possible since the execution speed of such loops may be more than a decimal order of magnitude higher than the one of conventionally translated loops. Unfortunately, the necessary dependence analysis is quite complicated, and often it cannot be decided whether it is possible to restructure a loop without affecting its semantics, in which case it must be executed sequentially. Therefore, a Fortran programmer has to carefully avoid any constructs within innermost loops which may inhibit vectorization. In fact, he must know *how* the specific compiler in use does vectorize. It would of course be better to use a suitable language construct instead. The features of Fortran 90 (see e.g. [Metcalf 1987]) reduce this problem a little bit. However, as will be shown in Section 1.4, Fortran 90 array expressions require dependence analysis, too.

This thesis introduces a new language called Oberon-V, based on the programming language Oberon [Wirth 1988a]. The principal new features of

Oberon-V are its ability to directly express potential parallelism in assignments and a construct to specify subarrays which may be passed as arguments to procedures. The former allows for efficient translation into vector instructions without dependence analysis, whereas the latter is an essential prerequisite for the programming of numerical applications (which was missing in Oberon's predecessors Modula-2 [Wirth 1985a] and Pascal [Wirth 1971]). The responsibility for the correctness of parallel executable program parts is delegated to the programmer. This decision is justified by practical examples of realistic programs.

The thesis is organized as follows: in the first two sections a hypothetical vector computer will be introduced, allowing to illustrate the basic concepts of such machines and serving as a basis for (vector) code examples. The problems arising when trying to vectorize DO loops are illustrated in Section 1.3. A special feature of Fortran 90, so-called array expressions, is investigated in Section 1.4, whereas Section 1.5 shows the problems of passing (sub-)arrays to subroutines and procedures. Chapter 2 describes the main concepts of Oberon-V; possible translation schemes are shown in Chapter 3. A survey of related programming languages may be found in Chapter 4. Chapter 5 describes the overall structure of a complete Oberon-V compiler called OV for the Cray Y-MP [Cray 1988]. Measurements made on a small collection of typical programs compiled with OV are arranged and discussed in Chapter 6. The thesis ends with the conclusions in Chapter 7. The Oberon-V language report may be found in Appendix A. Several typical programs are contained in Appendix B.

1.1 A Hypothetical Vector Computer

The topics of this thesis are programming languages and vector computers. While it is assumed that the reader has a certain understanding of what a programming language is, the notion of a vector computer might not be as clear. For the purpose of abstraction, a *hypothetical vector computer* is introduced that will be used instead of a real machine architecture that would have to be explained in full detail and with all its deficiencies. As a simple justification, it is stated that this machine reflects the basic *programming model* of many modern (register-to-register) vector processors, including machines such as Cray-1 [Russel 1978], Cray Y-MP [Cray 1988] and Cray-2, or NEC SX-2 and NEC SX-3 [NEC 1989]. In the following, the term vector computer always refers to this machine model, unless it is explicitly specified otherwise.

All vector computers share the concepts of *vectors* to which *vector instructions* can be applied. A vector is a finite sequence of scalar values called elements. It may be held either in a *vector register* or in memory. In the latter case, a vector is determined by its *starting address* c_0, a *stride* c_1 and its *length* l. The starting address is the address of the first element in memory whereas the stride specifies the (possibly negative) constant address difference between two consecutive elements in memory. The length is the number of vector elements (Figure 1.1).

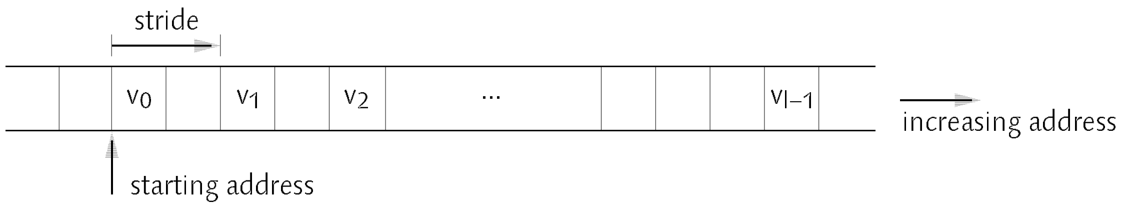


Figure 1.1 Memory Mapping of Vectors

In analogy to the notion of a *variable address*, the term *vector address* is used for the pair (c_0, c_1). Obviously, the set of addresses of all vector elements corresponds to the range of an *affine function* $f(x) = c_0 + c_1x$ with x restricted to the set {0, 1, ... l–1} (see also Section 3.2).

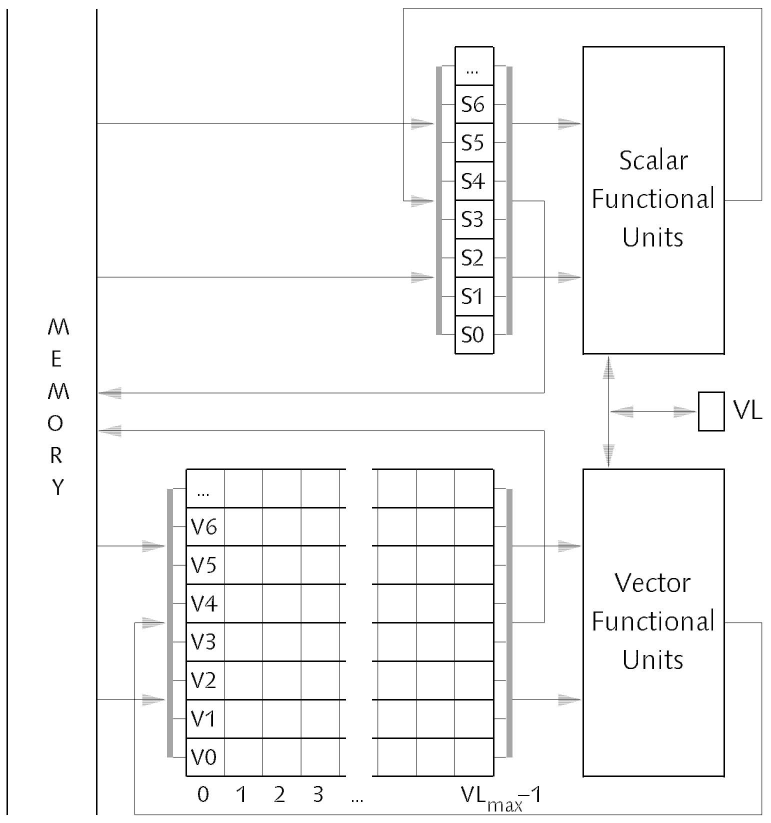


Figure 1.2 Hypothetical Vector Computer

The length of vectors held in main memory is virtually unlimited whereas vector registers may only hold vectors of a certain maximum length VL_{max}. Besides the registers for scalar values, a vector processor provides a special register file consisting of vector registers. For simplicity, it is assumed that both the scalar and the vector register file provide an unbounded number of registers; the former are denoted by S0, S1, etc., the latter by V0, V1, and so on (Figure 1.2).

The power of a vector computer lies in the *vector instructions*. A vector instruction is a machine instruction that performs an operation on an entire vector. It may be a vector load or store instruction to access (parts of) vectors in memory or scalar operations applied element-wise to their operand vector(s). A special *vector length register* VL is used to specify the number of elements to be manipulated by a vector instruction.

On a real machine, scalar and vector instructions are executed by the corresponding functional units. Since the execution of an instruction may take more than a single clock cycle (this is especially true for vector instructions), functional units are *segmented* and the instructions are executed in a *pipelined* fashion. Furthermore, if different functional units exist for different operations (e.g. an add and a multiply unit), different operations can be executed concurrently. It is this parallelity on the instruction level combined with the possibility to issue a sequence of operations by a single vector instruction that leads to the significant speedup of vector computers. However, timing aspects are not of relevance here. They will be discussed later. The hypothetical machine provides the following instruction classes:

Class	*Instruction*
scalar operations	Si := Sj op Sk
scalar comparisons	Si := Sj rel Sk
scalar load	Si := M[Sj + Sk]
scalar store	M[Sj + Sk] := Si
vector operations	Vi := Vj op Vk
vector comparisons	Si := Vj rel Vk
get vector element	Si := Vk[Sj]
set vector element	Vi[Sj] := Sk
vector load	Vi := M[Vj, Sk]
vector store	M[Vj, Sk] := Vi
set VL	VL := Sj
get VL	Si := VL
vector merge	Vi := Vj | Vk (Sl)
jumps	jump Sj (~Sk)

With the exception of jumps, the instructions are to be read as assignments. The letters i, j, k and l stand for a register number and the operand position; op denotes an arithmetic or logic operation and rel stands for one of the relations =, ≠, <, ≤, > or ≥. A Vj operand may be substituted by an Sj operand. Wherever an Sj operand is expected, an immediate value x (i.e. an integer constant) may be used instead. An Sk or Vk operand may be substituted by the immediate value 0. In the examples, a zero operand will be simply omitted. A scalar comparison "Si := Sj rel Sk" yields 1 or 0 depending on whether the relation is true or false.

The vector instructions deserve a more precise explanation: every instruction is executed for as many vector elements as specified by the current value of the vector length register VL. A vector operation of the form "Vi := Vj op Vk" (or "Vi := Sj op Vk", obtained by substitution of Vj by Sj) is to be understood as "Vi[e] := Vj[e] op Vk[e]" (or "Vi[e] := Sj op Vk[e]" respectively) for all values of e in the set {0, 1, ... VL–1}. A vector load instruction of the form "Vi := M[Vj, Sk]" is to be read as "Vi[e] := M[Vj[e] + Sk]"; hence the address of the element e is computed by the value of the element e of Vj plus the value of Sk. This is sometimes called a *vector gather* operation. A similar interpretation of the corresponding store operation leads to a *vector scatter* operation. By substituting the Vj operand with Sj, the conventional vector load/store instructions are obtained: e.g. "Vi := M[Sj, Sk]" is to be read as "Vi[e] := M[Sj∗e + Sk]"; i.e. Sk denotes the starting address of the vector whereas Sj denotes its stride. A constant stride is expressed by simply using an immediate operand instead of Sj. Hence, the pair (Sk, Sj) represents the *vector address*.

Vector comparisons "Si := Vj rel Vk" are used to compare entire vectors. The bit e of the destination register corresponds to the result of the comparison of vector element Vj[e] with vector element Vk[e]. If the comparison yields true, the corresponding bit is set to 1, otherwise it is set to 0. This instruction is useful for the implementation of conditional assignments. The vector merge instruction "Vi := Vj | Vk (Sl)" merges elements from Vj and Vk, depending on the vector mask in Sl: if bit e of Sl is set, Vi[e] becomes Vj[e] else Vi[e] becomes Vk[e].

A special rule governs the "VL := Sj" instruction: not the value of Sj is assigned to VL but ((Sj–1) MOD VL_{max}) + 1; i.e. VL becomes the value of Sj MOD VL_{max} if Sj is not a multiple of VL_{max}, else VL becomes VL_{max}. As will be seen, this is quite useful for the implementation of operations on vectors longer than VL_{max} elements. A similar instruction exists for Cray vector computers (cf. [Cray 1988]).

Jump instructions "jump Sj (~Sk)" depend on the negation of the truth value in Sk. If Sk is 0, the jump is executed, otherwise execution continues with the instruction immediately following the jump. By substituting Sk with 0, unconditional jumps are obtained.

1.2 Vectorization: an Example

From a programmer's point of view, a vector computer looks much like a conventional machine; the only difference is the availability of vector instructions using vectors as operands. However, as long as they are ignored, there is no difference at all. On the other hand, in order to take full advantage of the potential computing power of a vector computer, it is vital to make use of the vector instructions. In the early days of supercomputing, some machines have been used solely because of their superior scalar performance, today a Cray Y-MP is outperformed by much smaller and cheaper machines (e.g. an IBM RS/6000) as long as no vector instructions are used. A simple example shall help to get familiar with the possibilities offered by vector instructions and especially with the hypothetical vector computer introduced in the previous Section. Note that vector instructions operate logically *in parallel* on a vector of data. Hence, a translator trying to use vector instructions must recognize which parts of a program may be executed in parallel.

The task in the following is to scale all elements of an array A by a value x and to increment the scaled value by the corresponding element of an array B. Both arrays contain a constant number of elements (n). In Oberon [Wirth 1988a] this task may be formulated as follows:

```
CONST
  n = 64;
VAR
  A, B: ARRAY n OF REAL;
  x: REAL;
  i: INTEGER;
...
i := n;
WHILE i > 0 DO DEC(i); A[i] := A[i] * x + B[i] END
```

A translation of this example into scalar machine instructions is shown below. Index checks are omitted. Floating-point operations are distinguished from integer arithmetic by an "F" following the operation symbol. A semicolon introduces a comment ending at the end of the line.

```
;       S0                      address of A
;       S1                      address of B
;       S2                      variable x
;       S3                      index i
;
        S3 := 64                ; i := n
Loop    S10 := S3 > 0           ; i > 0
        jump Exit (~S10)        ; WHILE i > 0 DO
        S3 := –1 + S3           ; DEC(i)
|       S11 := M[S0 + S3]       ; A[i]
|       S12 := S2 *F S11        ; A[i] * x
|       S13 := M[S1 + S3]       ; B[i]
|       S14 := S12 +F S13       ; A[i] * x + B[i]
|       M[S0 + S3] := S14       ; A[i] := A[i] * x + B[i]
        jump Loop               ; END
Exit    ...
```

If possible, a compiler for a vector computer should generate vector instructions instead. Since in this example any two loop iterations i_1 and i_2 are *independent* of each other, i.e. it does not matter in what order they are executed (see Chapter 2), they may be executed in parallel. If n is not greater than VL_{max}, the entire loop can be translated into a sequence of a few vector instructions. Note the similarity between the loop body above (|) and the solution below:

```
;       S0                      address of A
;       S1                      address of B
;       S2                      variable x
;
        VL := n                 ; set vector length
|       V11 := M[1, S0]         ; A
|       V12 := S2 *F V11        ; A * x
|       V13 := M[1, S1]         ; B
|       V14 := V12 +F V13       ; A * x + B
|       M[1, S0] := V14         ; A := A * x + B
```

The elements of the arrays A and B are loaded from memory using vector load instructions: the vector to be accessed is determined by its starting address (which is the address of A or B respectively), its stride (which is 1 because it is assumed that the elements of both arrays are stored consecutively in memory) and the number of elements (which is implicitly determined by the contents of the VL register). The translation indicates that the entire loop could indeed be formulated in some form of a "vector assignment" such as "A := A*x + B" if it would be available in the programming language at hand.

Unfortunately, most vectors are much longer than the largest vector held in a single vector register or their lengths are not known at compile time, which requires a more sophisticated translation scheme. The straight-forward solution is to map an operation on a "long" vector to a sequence of operations on "short" vectors (*vector slicing*), i.e. to wrap up operations on short vectors in a loop. In the example above, this idea could lead to the following code:

```
;        S0                   address of A
;        S1                   address of B
;        S2                   variable x
;        S3                   counter c
;        S4                   slice pointer A↑
;        S5                   slice pointer B↑
;
         S3 := n              ; initialize counter
         S4 := S0             ; initialize A↑
         S5 := S1             ; initialize B↑
Loop     S10 := S3 > 0        ; c > 0
         jump Exit (~S10)     ; WHILE c > 0 DO
         VL := S3             ; set vector length
|        V11 := M[1, S4]      ; A[i]
|        V12 := S2 ∗F V11     ; A[i] ∗ x
|        V13 := M[1, S5]      ; B[i]
|        V14 := V12 +F V13    ; A[i] ∗ x + B[i]
|        M[1, S4] := V14      ; A[i] := A[i] ∗ x + B[i]
         S15 := VL            ; get vector length of this iteration
         S3 := S3 – S15       ; c := c – VL
         S4 := S4 + S15       ; A↑ := A↑ + VL
         S5 := S5 + S15       ; B↑ := B↑ + VL
         jump Loop            ; END
Exit     ...
```

For each long vector to be sliced (i.e. the arrays A and B in the example above), an additional register denoting the starting address of the current vector slice is introduced: registers S4 and S5 point to the slices A↑ or B↑ respectively. Initially they are set to the array base addresses. Then they are incremented in each loop iteration by the length of the current vector slice. The loop terminates if the value of a counter register (S3) drops to zero (Figure 1.3).

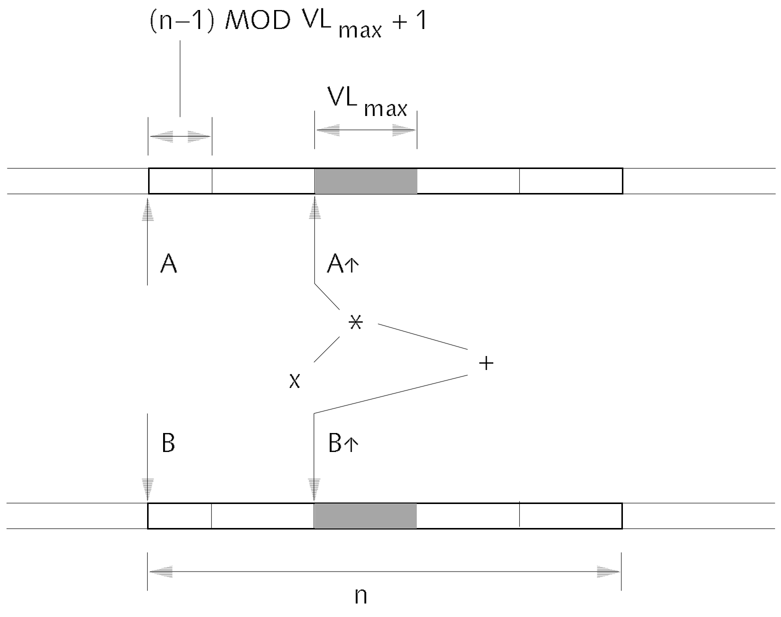


Figure 1.3 Vector Slicing

A subtle point is the computation of the current vector length: in order to process as many elements as possible within a single iteration, the vector length register has to be set to the maximum length possible, i.e. to VL_{max}. Unfortunately, the length of a long vector is in general not a multiple of VL_{max}. Hence, some kind of "start-up" (or "shut-down") code is necessary to correctly implement a long vector operation. However, using the semantics of the "VL := Sj" instruction (cf. Section 1.1), no specific code is necessary. The reader may see himself that this method leads to the correct number of loop iterations and vector elements to be processed.

1.3 Vectorization of Fortran DO Loops

Originally introduced by J. Backus in 1954 as the first "high-level" language, Fortran is still the mainstream programming language for numerical applications. Fortran has survived several standardization processes; the latest ANSI standard is called Fortran 90 (cf. [Metcalf 1987]) and evolved from Fortran 77 [Brainerd et al. 1978] which in the following is simply referred to as Fortran. In this section it is not speculated on why Fortran has been so successful over the last 40 years, but it is concentrated on its suitability for the programming of vector computers.

In fact, Fortran does not support the programming of vector computers by means of special language constructs or data types. In order to make effective use of such a machine, a Fortran compiler therefore has to analyze and restructure a conventional program so that vector instructions can be generated. The most promising program parts for such an attempt are innermost DO loops. It should be clear from the previous examples (Section 1.2) that many loops may be vectorized, however not all. Whenever some form of *dependence* between different loop iterations exist, vectorization may be inhibited. As a simple example let us consider the following DO loop:

```
      DO 10 I = 1, 10
         A(I) = B(5*I – 1)          (S1)
         B(3*I + 10) = C(I)         (S2)
10    CONTINUE
```

A, B and C are arrays. The two assignment statements are denoted by S_1 and S_2 respectively, and a specific *instance* of an assignment is denoted by $S_k(i)$, i.e. $S_k(i)$ denotes the assignment S_k in the i-th iteration, and k = 1, 2. If the loop is executed sequentially, the statement S_1 may be (flow-)dependent on S_2 because the expression 3*I + 10 in S_2 may assume the same value for a given I = i_2 as the expression 5*I – 1 in S_1 for an I = i_1 in a later iteration (i.e. $i_1 > i_2$). However, it is not immediately clear that such a dependence exists. Because the example is so small, it is possible to consider the values of *all* indices for B in S_1 and S_2:

Index values for B in S_1:	4, 9, 14, **19**, 24, 29, **34**, 39, 44, 49
Index values for B in S_2:	13, 16, **19**, 22, 25, 28, 31, **34**, 37, 40

There are the same indices for I = 4 in S_1 and I = 3 in S_2 (19) and again for I = 7 in S_1 and I = 8 in S_2 (34). Thus, $S_1(4)$ is flow-dependent on $S_2(3)$, i.e. $S_1(4)$ must be executed *after* $S_2(3)$. It is obvious that a parallel execution of S_1 for all index values (using vector instructions) followed by a parallel execution of S_2 would change the effect of the loop.

If, still in the same example, the index expression for B in S_2 would be 3*I + 9 instead of 3*I + 10, one would obtain the index sequence 12, 15, 18, 21, 24, 27, 30, 33, 36, 39. In this case, the same index value (24) would be assumed only once for I = 5 in both statements. The reader may see himself that the suggested parallel execution would be allowed in this case and the effect of the entire DO loop would not be changed.

In general the situation is not as clear because the index range is not always as small or even unknown. Nevertheless, there are better, i.e. analytical techniques to detect dependences. The example shows the following relationships: an instance $S_1(i_1)$ uses a value computed by an instance $S_2(i_2)$ iff an iteration I = i_1 follows an iteration I = i_2 and the variables denoted by B(5*I – 1) in S_1 and B(3*I + 10) in S_2 are identical. More formally, iff two integers i_1 and i_2 exist so that the following (in-)equalities hold

$$(5i_1 - 1 = 3i_2 + 10) \wedge (i_1 > i_2) \wedge (1 \le i_1, i_2 \le 10)$$

then $S_1(i_1)$ uses a value computed by $S_2(i_2)$. The first equation can be written as $5i_1 - 3i_2 = 9$ which is a linear diophantine equation in two integer variables. In order to have a dependence, i_1 must be greater than i_2 and since both i_1 and i_2 are values of the index (or induction) variable I, they must also satisfy $1 \le i_1, i_2 \le 10$.

In general, dependence analysis is complex and may require the solution of linear diophantine equations in several variables with additional constraints. However, even if techniques for solving such inequalities were available (such methods do exist, see below), they would not always allow to decide in every case whether a dependence exists or not. Let us consider a slightly modified example:

```
        DO 10 I = 1, 10
           A(I) = B(c1*I – 1)           (S1)
           B(c2*I + 10) = C(I)          (S2)
10      CONTINUE
```

The constants 5 and 3 in the index expressions of B have been substituted by the variables c_1 and c_2. If it is not possible for a compiler to determine the actual values of c_1 and c_2 at compile time, there is little to say about the solution(s) of a corresponding diophantine equation. Worst-case assumptions must then be taken into account and a compiler must generate sequential code even if the programmer knows that no dependence exists. The situation is similar, if the index expressions are more complex or if they involve an index array:

```
        DO 10 I = 1, 10
           A(I) = A(111 – I**2)         (S1)
           B(I) = B(C(I))               (S2)
10      CONTINUE
```

Although both index expressions I and 111 – I**2 in S_1 denote disjoint sets of indices, a compiler in general cannot recognize this situation because the right-hand expression is not linear in I. For the second statement, a compiler would have to decide whether there are two integers i_1 and i_2 with $i_1 \ne i_2$ and $1 \le i_1, i_2 \le 10$ such that $i_1 = C(i_2)$ (or whether C(i) = i for all i's), which is in fact impossible since in general the contents of C are unknown to a compiler. However, a programmer may know that all instances of S_1 and S_2 are independent of each other, and hence a parallel execution would be possible:

```
;        S0                  address of A
;        S1                  address of B
;        S2                  address of C
;        S3                  address of integer vector 1, 2, ... VLmax
;
         VL := 10            ; set vector length
         V10 := M[1, S3]     ; load the integers I = 1, 2, ... 10
         V11 := V10 * V10    ; I**2
         V12 := 111 – V11    ; 111 – I**2
         V13 := M[V12, S0]   ; A(111 – I**2)
         M[1, S0] := V13     ; A(I) = A(111 – I**2)
         V14 := M[1, S2]     ; C(I)
         V15 := M[V14, S1]   ; B(C(I))
         M[1, S1] := V15     ; B(I) = B(C(I))
```

Up to now, there have been considered relatively simple loops containing assignment statements only. A compiler can generate vector instructions for them if no dependences exist that inhibit vectorization. The necessary dependence analyses are difficult and in general worst-case assumptions must prevail. Powerful techniques exist today that enable a compiler to vectorize many cases of typical loops, including certain loops containing IF statements. The interested reader is referred to the literature; e.g. [Lamport 1974], [Allen 1987], [Burke 1986]. A thorough introduction into DO loop dependence analysis and further references can be found in [Banerjee 1988].

The main advantage of having a compiler that is able to vectorize ordinary DO loops is that an immense amount of old software may profit immediately from vector computers. However, optimal performance depends on more than only a few vectorizable loops. An important role is played by memory access patterns (cf. Chapter 6). A programmer should therefore always have in mind that he is programming a vector computer. Furthermore, the use of arbitrary statements in a loop usually inhibits vectorization (an entire chapter of the Cray Fortran Reference Manual is dedicated to such exceptions [Cray 1986]). On the other hand, some frequently used code patterns such as reduction loops (e.g. for dot product) are especially recognized by the compiler and then translated into a specific code sequence although in general such loops do not vectorize (see also [Cray 1986]). Again, the programmer must exactly know the applicable code pattern.

1.4 Vectorization of Fortran 90 Array Expressions

Fortran 90 explicitly supports *array expressions*, i.e. expressions where the operands may be entire arrays [Metcalf 1987]. The involved arrays must have the same *shape*, i.e. they must have the same number of dimensions and corresponding dimensions must have the same length. Operations are then applied element-wise. The array expression

```
REAL, ARRAY (20, 30) :: A, B, C
...
A := B + C * 0.5
```

corresponds to the following DO loop nest (using the new Fortran 90 syntax):

```
DO I = 1, 20
   DO J = 1, 30
      A(I, J) = B(I, J) + C(I, J) * 0.5
   END DO
END DO
```

The definition of the semantics of array expressions requires the complete evaluation of the expression on the right-hand side of an assignment before any (partial) result is assigned to the array variable on the left-hand side. Hence, the straight-forward translation of an array expression into nested DO loops in general is only correct if the array on the left-hand side is "disjoint" from all arrays in the expression on the right-hand side of the assignment (i.e. if the sets of the array elements are disjoint) or equal to those to which it is not disjoint. If this condition holds, an array expression can also be translated into vector instructions without further dependence analysis. If this condition is not fulfilled, an auxiliary array is usually required to correctly implement the array assignment.

Note that the introduction of such an auxiliary array (by the compiler!) may have a significant impact on the efficiency of the assignment since the array must be allocated first (its size may not be determinable at compile time; see below) and after the evaluation of the entire array expression and the assignment to the auxiliary array, the auxiliary array must be copied to the destination array of the assignment. Finally, it must be deallocated. Thus, a compiler should try to avoid its introduction whenever possible. As already explained, this decision requires a "disjoint-or-equal" test. Because not only entire arrays but also subarrays may be used in array expressions, this test might not be decidable at compile time (the subarrays might be specified by variables). The array assignment

```
A(1 : 100, K) = A(J : J+99, K)
```

can only be translated into a simple loop (and hence into vector instructions)

```
DO I = 1, 100
   A(I, K) = A(J + I – 1, K)
END DO
```

if the subarrays A(1 : 100, K) and A(J : J+99, K) are disjoint or equal (in case of overlapping subarrays a translation into a simple loop or vector instructions respectively may also be possible; however, in this case the "iteration direction" may be not clear at compile time). If the value of J is unknown at compile time, a pessimistic translation using a temporary array is usually unavoidable (or a run-time check is required to select between several different code sequences). Unfortunately, the situation is even worse: in a Fortran 90 subarray specification it is not only possible to specify a lower and an upper bound, but also a stride. For example, A(1 : 99 : 2, K) denotes the elements A(1, K), A(3, K), ... A(99, K). In this case, arrays involved in an array expression are not simply rectangular subarrays but consist of "grid elements" of the subscripted arrays. If one of the three components of an array subscript (begin, end, stride) is a variable, the length of the array is unknown at compile time and must be determined at run time (possibly involving division). The access order of the elements depends on the sign of the stride. If the same array name occurs on both sides of an array assignment, the decision whether its subscripts denote disjoint or equal array elements may lead to diophantine equations.

Although at first sight, Fortran 90 array expressions seem to be an elegant construct to support vector or parallel computers, their implicit complexity requires a significant amount of work that is similar to the amount of work required for the vectorization of loops. Furthermore, since most of this work is hidden from the programmer, he might not even be aware of the heavy-weight tool he is using.

1.5 Array Parameters in Fortran

In Fortran an array is always passed by reference and the programmer is responsible for its correct usage within the subroutine. If the length of the array is unknown at compile time, it is passed as an additional parameter *by convention*. Within the subroutine, the length parameter is then used to correctly define the array as a local variable. If an array is multi-dimensional, so-called *leading dimensions* (LDx) are passed as additional parameters. The matrix multiply subroutine

```
SUBROUTINE MUL (A, LDA, B, LDB, C, LDC, N)
INTEGER LDA, LDB, LDC, N
REAL A(LDA, 1), B(LDB, 1), C(LDC, 1)
...
END
```

expects three N x N arrays A, B and C. The leading dimensions of these arrays LDA, LDB, and LDC and the size N are passed explicitly. Within the subroutine, the leading dimensions are used to declare the arrays A, B and C. In fact, this is only a specification of the "address computation rule" for the compiler since the length of the last dimension is often simply set to 1 (and no index checks are performed in Fortran). A correct call of the subroutine for three 100 x 100 arrays X, Y, Z would be the following

```
REAL X(100, 100), Y(100, 100), Z(100, 100)
...
CALL MUL(X, 100, Y, 100, Z, 100, 100)
```

Since for the parameters A, B and C simply the addresses of the corresponding arrays X, Y, and Z are passed, arbitrary subarrays may be specified as arguments. The following call

```
CALL MUL(X(10, 20), 100, Y(30, 40), 100, Z(50, 60), 100, 10)
```

performs a matrix multiplication on the 10 x 10 subarrays X(10 : 19, 20 : 29), Y(30 : 39, 40 : 49) and Z(50 : 59, 60 : 69) (using the Fortran 90 notation). If the leading dimension arguments are modified, even worse tricks are possible. Note that these "techniques" are the only methods to pass subarrays in Fortran. Therefore they must also be used in high-quality numerical applications, such as Linpack [Dongarra et al. 1979].

Fortran 90 allows the explicit specification of subarrays as explained in Section 1.4. Within subroutines, an array parameter may be declared to assume the size of the corresponding argument. Using a RESHAPE transformation, the shape of the array may be arbitrarily changed and then passed as an argument. However, the old-style parameter passing conventions are still valid.

2 A Language Proposal

In the previous chapter it has been argued, why conventionally used programming languages such as Fortran 77 are not sufficient for the task of programming vector computers. It has been concentrated on two points, namely vectorization of loops and parameter passing of subarrays. For a more thorough survey of related languages, see Chapter 4.

Two new language constructs are proposed which subsequently are shown to solve these problems. These constructs have been intergrated into an experimental language called Oberon-V. Oberon-V evolved from Oberon which was chosen as a basis because it is one of the very few modern general-purpose programming languages which fulfill the main criteria for good language design: *simplicity*, *security*, *fast translation*, *efficient object code* and *readability* [Hoare 1974].

In the following sections only the new language constructs are explained; the full language report may be found in Appendix A. Section 2.3 comprises a discussion of design decisions which lead to differences between Oberon and Oberon-V. For a general introduction to Oberon the reader is referred to the literature [Reiser 1992].

2.1 The ALL Statement

FOR loops and similar loop constructs in other languages (e.g. Fortran DO loops) explicitly describe a *deterministic* and *sequential* iteration process. In many cases the task to be performed is *overspecified* by such loop constructs. Let us consider a simple example: the task is to scale all elements of an array A. A FOR loop such as the following

```
VAR
   A: ARRAY 100 OF REAL;
   c: REAL;
...
FOR i := 0 TO 99 DO
   A[i] := A[i] * c
END
```

specifies not only that each element of A be scaled with c but also in which order they are to be scaled. Indeed, the exact execution order does not matter in the example and thus could be *non-deterministic*. Furthermore, even the sequential execution of the loop is unimportant, since for each value of i different (and thus independent) variables are accessed. In the example, several or all elements could also be scaled in *parallel* without changing the effect of the FOR loop.

It is interesting that the informal description of the task "multiply all elements of array A with the scale factor c" does *neither specify a particular order nor sequential execution*. It is the programmer (forced by the programming language he uses) who translates the task into a deterministic and sequential process. Thus, inherent properties of the task are *destroyed* by "doing more than necessary" during its translation into a program. In order to vectorize such a loop, a compiler has to *recover* these properties using dependence analysis.

One must carefully distinguish between non-deterministic and parallel execution. If there were an imaginary FOREACH statement which would allow to express the *sequential execution* of a statement sequence in a *non-deterministic order*, it would be possible to compute the sum s of all elements of an array A by

```
VAR
   A: ARRAY 100 OF INTEGER;
   s: INTEGER;
...
s := 0;
FOREACH i IN {0 .. 99} DO
   s := s + A[i]
END
```

since the order in which the elements are added does not matter (at least for integers). Obviously, parallel execution would lead to a wrong result. The importance of having such non-deterministic language constructs has been stressed before; they do not only allow to avoid overspecification of programs but as a consequence also simplify program verification. Examples are the non-deterministic evaluation of guards in Dijkstra's IF and DO statement

[Dijkstra 1976] or the programming notation UNITY used for the development of parallel programs [Chandy 1988].

In Oberon-V a new structured statement called *ALL statement* has been introduced. The ALL statement is used to specify the independent execution of a sequence of assignments for a set of (index) values within specified intervals. The assignment sequence is prefixed by a *range declaration* which is used to specify these intervals or *ranges*. A similar construct (FORALL) may be found in Vienna Fortran [Zima 1992]; a Fortran offspring especially directed towards programming of parallel machines. As an introductory example let us consider the ALL statement corresponding to the scaling example:

```
VAR
   A: ARRAY 100 OF REAL;
   c: REAL;
...
ALL r = 0 .. 99 DO
   A[r] := A[r] * c    (* S(r) *)
END
```

The range declaration r = 0 .. 99 specifies a new *range identifier* r which is associated with the range 0 .. 99. Within the ALL statement, the range identifier r stands for any integer value i within its associated range, i.e. for any of the integers 0, 1, 2, ... 99 in the example. Execution of the ALL statement means that the enclosed assignment sequence S(r) is executed *exactly once* for each value i that can be assumed by the range identifier r. The order in which r assumes such a value i is *undefined.* Furthermore, in order to be correct, the ALL statement requires that different assignment sequences $S(i_1)$ and $S(i_2)$ with $i_1 \neq i_2$ be *independent* of each other; i.e. that no variable accessed or modified in the assignment sequence where $r = i_1$ is modified in any other assignment sequence where $r = i_2$. In the example, the variables accessed or modified by $S(i_1)$ obviously are different from the variables accessed by $S(i_2)$ for $i_1 \neq i_2$, thus any two assignment sequences $S(i_1)$ and $S(i_2)$ are independent of each other. Because they are independent, they may even be executed in parallel.

In general, an ALL statement may introduce more than a single range identifier. In this case, the enclosed assignment sequence is executed exactly once for each combination of values the range identifiers may assume and a similar independence rule must hold. Using EBNF, the syntax of the general ALL statement is:

```
AllStatement = ALL RangeDeclaration DO AssignmentSequence END.
AssignmentSequence = [Assignment {";" Assignment}].
RangeDeclaration = RangeList {"," RangeList}.
RangeList = ident {"," ident} "=" Range.
Range = Expression ".." Expression.
```

(for the definition of the non-terminal symbols *Assignment* and *Expression* as well as the symbol *ident* see Appendix A). The semantics of the ALL statement can be specified by a *rule of inference* (cf. Section 2.3.6) which also serves as a proof outline for verifying programs containing ALL statements. For brevity, only the case with a single range identifier is shown here (the general definition is again found in Appendix A). P, Q, P(r) and Q(r) denote predicates describing the program states before and after the execution of the ALL statement or the enclosed assignment sequence S(r), respectively:

$\{P\}$ ALL $r = a \,..\, b$ DO $S(r)$ END $\{Q\}$

holds if conditions P(r) and Q(r) exist, such that

(1) $P \Rightarrow (\forall i:\ a \le i \le b:\ P(i))$
(2) $\forall i:\ a \le i \le b:\ \{P(i)\}\ S(i)\ \{Q(i)\}$
(3) $(\forall i:\ a \le i \le b: Q(i)) \Rightarrow Q$

and

$\forall i, j:\ (a \le i, j \le b) \wedge (i \ne j):\ (in(i) \cap out(j) = \emptyset) \wedge (out(i) \cap out(j) = \emptyset)$

with $in(i) \equiv$ set of variables that have been accessed by S(i)
and $out(i) \equiv$ set of variables to which values have been assigned to by S(i)

When applied as a proof scheme to the scaling example one obtains:

```
{$P \equiv (\forall i:\ 0 \le i \le 99:\ A[i] = A_0[i])$}
ALL r = 0 .. 99 DO
   {$P(r) \equiv (A[r] = A_0[r])$}   A[r] := A[r] * c   {$Q(r) \equiv (A[r] = A_0[r] * c)$}
END
{$Q \equiv (\forall i:\ 0 \le i \le 99:\ A[i] = A_0[i] * c)$}
```

It is easily verifyed that (1), (2) and (3) hold. Furthermore, all assignments are independent since $in(i) = out(i) = \{A[i]\}$ and of course

$$\forall\, i, j : (0 \le i, j \le 99) \wedge (i \ne j) : \{A[i]\} \cap \{A[j]\} = \emptyset$$

Thus, the example using the ALL statement correctly implements the scaling of array A. Since all assignment sequences must be independent and thus can be regarded "separately", proofs for larger ALL statements are in general not much more complicated. Note that it is *necessary* to check whether independence holds or not, as will be shown by the following counter example (in particular, independence is *not implied* by (1), (2) and (3)):

```
{P ≡ (s = 0)}
ALL r = 0 .. 99 DO
   {P(r) ≡ (s = 0)}    s := s + 1    {Q(r) ≡ (s = 1)}
END
{Q ≡ (s = 1)}
```

Obviously (1), (2) and (3) hold, but in(i) $\cap$ out(j) = {s} $\ne \emptyset$ for different i and j. Because the ALL statement may be either executed sequentially or in parallel (or even partially sequential and partially in parallel), the result is indeed undefined.

If a particular sequential execution order were specified, an ALL statement introducing n range identifiers

```
ALL $r_1$ = $a_1$ .. $b_1$, $r_2$ = $a_2$ .. $b_2$, ... $r_n$ = $a_n$ .. $b_n$ DO
   S($r_1$, $r_2$, ... $r_n$)
END
```

where S denotes an assignment sequence containing the range identifiers r_1, r_2, ... r_n could be implemented by a loop nest

```
$i_1$ := $a_1$;
WHILE $i_1$ <= $b_1$ DO
   $i_2$ := $a_2$;
   WHILE $i_2$ <= $b_2$ DO
      ...
         $i_n$ := $a_n$;
         WHILE $i_n$ <= $b_n$ DO
            S($i_1$, $i_2$, ... $i_n$);
            INC($i_n$)
         END;
      ...
      INC($i_2$)
   END;
   INC($i_1$)
END
```

Such a conversion must be understood as overspecification of a task, for which an ALL statement would suffice. It immediately follows that the reverse conversion is wrong in general.

Due to its properties, an ALL statement can (almost) always be translated into vector instructions (if an assignment contains boolean operators or function calls, vectorization may be inhibited, see Section 3.7.2). Hence, it is easily possible to decide, whether certain parts of a program vectorize or not. This stands in sharp contrast to the situation in Fortran.

Instead of demonstrating more elaborate examples underlining the usefulness of ALL statements for practical programs, the reader is referred to Appendix B where he may find a few longer program samples extensively using ALL statements.

2.2 Array Constructors

Oberon-V array constructors are used to *construct* new arrays *consisting of elements of other arrays*. Note the difference between Oberon-V array constructors and constructors in other languages which are used to construct arrays or records of arbitrary components (e.g. Fortran 90 array constructors [Metcalf 1987]). An Oberon-V array constructor consists of a range declaration similar to the one used in ALL statements, followed by an expression using the range identifiers which have been declared in the range declaration. The syntax of the array constructor is:

```
ArrayConstructor = "[" RangeDeclaration ":" Expression "]".
RangeDeclaration = RangeList {"," RangeList}.
RangeList = ident {"," ident} "=" Range.
Range = Expression ".." Expression.
```

(for the definition of the non-terminal symbol *Expression* and the symbol *ident* the reader is again referred to Appendix A). An Oberon-V array constructor is of the form

$[r_1 = a_1 .. b_1, r_2 = a_2 .. b_2, ... r_n = a_n .. b_n: E(r_1, r_2, ... r_n)]$

where $E(r_1, r_2, ... r_n)$ is an expression possibly containing the range identifiers r_1, r_2, ... r_n. Within the expression, each range identifier r_k stands for any integer value i_k within the range associated to r_k as in ALL statements, i.e. $i_k \in \{a_k, a_k + 1, ... b_k\}$. Thus, the array constructed by the array constructor denotes an (at least) n-dimensional array consisting of the elements E(i_1, i_2, ... i_n). Using the variables A: ARRAY 100 OF REAL and B: ARRAY 100, 100 OF REAL, a few examples shall illustrate legal array constructors and the array they construct:

Array Constructor	*Elements of the Constructed Array*	*Dimensions*
[r = 10 .. 20: A[r]]	A[10], A[11], ... A[20]	1
[r = 0 .. 10: A[3∗r + k]]	A[k], A[3+k], A[6+k], ... A[30+k]	1
[r = 0 .. LEN(B)–1: B[r, r]]	Diagonal of B	1
[r = 0 .. 99: B[99–r]]	B[99], B[98], ... B[0]	2
[r, s = 0 .. 99: B[s, r]]	Transpose of B	2
[r = 10 .. 20: ABS(A[r])]	ABS(A[10]), ABS(A[11]), ... ABS(A[20])	1
[r = 0 .. 9: A[r] ∗ B[r, r]]	A[0]∗B[0, 0], A[1]∗B[1, 1], ... A[9]∗B[9, 9]	1
[r = 0 .. 4: A[r] + A[r+5]]	A[0]+A[5], A[1]+A[6], ... A[4]+A[9]	1
[r, s = 0 .. 99: –B[r, s]]	–B	2
[r, s = 1 .. 10: 1/(r+s–1)]	10 x 10 Hilbert matrix	2

The first 5 examples construct *array variables*, since the expression following the range declaration in the array constructor consists of a *designato*r (see Appendix A.10.3) denoting a *variable*. In contrast, the last 5 examples construct *array values*, since the range declarations are followed by an arbitrary expression which is not only a designator denoting a variable. The right-most column denotes the number of dimensions of the constructed array (for an exact definition of "number of dimensions" see Appendix A.6.2). Often, the number of range identifiers corresponds to the number of dimensions of the constructed array; however, this is only true if the type of the expression is not an array (see 4-th example from top).

In Oberon-V, constructed *array variables* may be used as *arguments for open array variable parameters*. In order to make them efficiently and safely implementable, a few restrictions must apply, e.g. the subscripts of a designator must contain only "linear" range expressions. Thus, an array variable constructor [r = 0 .. 9: A[r∗r]] is not allowed. For a justification of these restrictions see Section 3.9; for an exact specification of array constructors and the restrictions see Appendix A.10.5 and A10.6.

Figure 2.1 illustrates the passing of a diagonal of the array A as an argument to the procedure *Scale*. Since the array constructor introduces a single range identifier r and the designator A[r, r+1] denotes a (zero-dimensional) *scalar variable*, the array constructor denotes a (1 + 0 =) 1-dimensional *subarray variable* of A. Within the procedure Scale, this subarray is simply referred to by the parameter a; element a[0] stands for A[0, 1], element a[1] stands for A[1, 2] and so on.

```
VAR
  A: ARRAY 10, 10 OF REAL;

PROCEDURE Scale (VAR a: ARRAY OF REAL; c: REAL);
BEGIN ALL i = 0 .. LEN(a) – 1 DO a[i] := a[i] * c END
END Scale;


...
Scale([r = 0 .. 8: A[r, r+1]], 3.14)
```

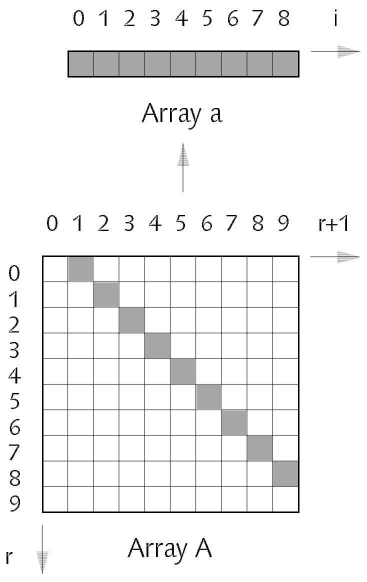


Figure 2.1 Subarray Passing

Since in Oberon-V no value parameters of structured type are allowed (see Section 2.3.4), *array value constructors* cannot be used as arguments in general, but only as arguments for the predefined Oberon-V functions SUM and PROD. These so-called *reduction functions* reduce the contents of an array to its sum or product respectively. They have been introduced, because they are used frequently (especially SUM for dot product) and because they can be implemented much more efficiently using special code patterns than using ALL statements and special computation tricks (cf. Chapter 6). In general, Fortran compilers also cannot vectorize DO loops implementing reduction functions. However, they recognize some special code patterns referring to obvious scalar implementations of SUM and PROD, and replace them by SUM and PROD respectively [Cray 1986].

The following example illustrates the use of the SUM reduction function to compute the matrix product of the matrices A and B with result C. For more elaborate examples, the reader is again referred to Appendix B.

```
PROCEDURE Mul (VAR A, B, C: ARRAY OF ARRAY OF REAL);
  VAR i, j: INTEGER;
BEGIN
```

```
      ASSERT(LEN(A, 1) = LEN(B));
      i := 0;
      WHILE i < LEN(A) DO j := 0;
         WHILE j < LEN(B, 1) DO
            C[i, j] := SUM([k = 0 .. LEN(A, 1)-1: A[i, k] * B[k, j]]);
            INC(j)
         END;
         INC(i)
      END
   END Mul;
```

2.3 Differences between Oberon and Oberon-V

In this section a few more subtle design decision are discussed which lead to differences between Oberon and Oberon-V. While it was felt that some features of Oberon are not essential for programming of numerical applications and hence were omitted to obtain a simpler design and implementation (e.g. certain basic types and control constructs have been omitted), other changes have been motivated in order to "repair" minor deficiencies (e.g. the different handling of procedure types). The latter can be understood as a modest criticism of Oberon. However, it is emphasized that Oberon-V must be considered as an *experimental language*, implemented mainly to prove the feasibility of a few specific concepts. Thus, whereas Oberon has already been used to program many applications ranging from simple text editors to compilers and even operating systems, and thereby proved to be appropriate for these tasks [Wirth 1988b], an equivalent suitability test for Oberon-V has not been performed.

2.3.1 Basic Types

Oberon provides a set of eight basic types. The numeric types constituted by the integer and real types form an *inclusion hierarchy*, i.e. the smaller type includes the larger type:

$$\text{SHORTINT} \subseteq \text{INTEGER} \subseteq \text{LONGINT} \subseteq \text{REAL} \subseteq \text{LONGREAL}$$

Due to this hierarchy, the conversion of a smaller type into the larger type can be automatically performed by the compiler, whereas the reverse conversion (e.g. LONGINT to INTEGER) requires the use of a type conversion function (SHORT), since the programmer has to ensure that the conversion is legal, i.e.

that the value of an expression of the larger type is also comprised by the smaller type. The main reason for having several integer or real types is storage economy [Wirth 1990].

In general, the use of different integer or real types within a single expression is not recommended: in such "mixed-type" expressions, determining the exact operations performed requires a careful and toilsome analysis of the expression. Often additional LONG or SHORT functions are required to guarantee the desired accuracy of the computation. If the type of a variable is changed to the next smaller type (e.g. from LONGINT to INTEGER) now unnecessary SHORT operations may remain undetected and lead to wrong results. As experience shows, such errors are difficult to locate, especially if no hard- or software support is available (e.g. overflow or range checks).

At least for integers, the problems disappear if only a single integer type, preferably the largest one, is used. Indeed, on modern RISC computers, different integer types are only treated differently in the implementation as far as memory accesses are concerned [Brandis et al. 1992]; e.g. once a SHORTINT is loaded into a (32-bit) register, all operations on it are essentially LONGINT operations. Thus, it would be more adequate to specify the "storage size" of a particular integer variable instead of specifying its type to be a "smaller integer" which then influences all expressions where the variable is used. Since storage economy is only a problem when having arrays (or records) containing integer components, it would suffice to have a possibility to specify the size of these integer components. Unfortunately, a similar solution for real types would not be appropriate, since a "long" real number cannot simply be shortened without loosing its numerical accuracy.

These ideas have not been pursued further. For simplicity, Oberon-V provides only a single integer and real type (INTEGER, REAL) but also the type COMPLEX. Together they form the inclusion hierarchy:

$$\text{INTEGER} \subset \text{REAL} \subset \text{COMPLEX}$$

The real and imaginary components of a complex number can be retrieved by two predefined functions RE and IM respectively. Conversion of real numbers to integers is possible with the predefined functions FLOOR, CEILING and TRUNC (see Appendix A.8.2).

2.3.2 Pointer Types

In Oberon the extension relationship defined over records is inherited by its corresponding pointer types; i.e. if a record R1 is an extension of a record R, then also a pointer type P1 with pointer base type R1 is an extension of a pointer type P pointing to R. Since a (record) type R1 extends a type R if it is equal to R or if is a direct extension of an extension of R ([Wirth 1988a], 6.3 Record Types), a pointer type P with pointer base type R is an extension of any other (!) pointer type Q with the same base type:

```
TYPE
   P = POINTER TO R;
   Q = POINTER TO R;
   R = RECORD ... END;

VAR
   p: P;
   q: Q;
...
p := q; q := p
```

Since R is the (zero) extension of itself, P can be regarded as an extension of Q. Vice versa, Q can be regarded as an extension of P. Due to the assignment rules for type extensions, it is possible both to assign p to q and q to p, without needing their types being equal. Indeed, P and Q are both extensions of each other and therefore should be considered as being equal. On the other hand, because the extension relation is only defined over records and its associated pointer types, the same game does not work anymore if R is an array type. Thus, the programmer is confronted with the strange fact that any two *different* pointer types pointing to the same record type are *de facto* equal types, but if they would point to an array they were not.

In Oberon-V, two pointer types are defined to be equal, if their base types are equal. Furthermore, a pointer variable of type P may be assigned to any other pointer variable of type Q, if the base type of P is an extension of the base type of Q. This definition eliminates the unpleasant situation explained above and it makes the expansion of the extension relationship to pointers superfluous.

In a second step, the reserved words POINTER TO have been replaced by a Pascal style pointer declaration using an arrow (↑) instead. This notation must be understood under consideration of the typical use of pointers. In Oberon, usually a named pointer type is declared together with its (named) base type. As a matter of style, having a pointer type called P, the corresponding base type identifier is built by appending some suffix, e.g. "desc". A dual convention is to call the base type B and the associated pointer type Bptr (by appending the suffix "ptr"). Frequently, only the pointer type is used in variable and parameter declarations, i.e. the name of the record type merely clutters the definition. However, type extensions require the name of the record type and thus prevent the use of anonymous record types in conjunction with a pointer type declaration as in P = POINTER TO RECORD next: P END.

Instead of relying on individual naming conventions, it is better to completely avoid the need for such conventions. Due to the Oberon-V compatibility rules for pointer types, it suffices to declare the pointer base types only. Whenever an associated pointer type is used, it is newly declared by writing the ↑ symbol followed by the base type name. The ↑ symbol then explicitly emphasizes the declaration of a pointer which enhances the readability of the program and simultaneously serves as some kind of naming convention. Of course, the same could have been achieved using the old POINTER TO notation, but then declarations would have become rather lengthy. For some examples demonstrating the typical usage of the Oberon-V pointer notation, the reader is referred to Section 5.2.

2.3.3 Procedure Types

Two types designated by two identifiers T1 and T2 are identical in Oberon if both identifiers T1 and T2 are equal, if they are declared to be identical in a type declaration of the form T1 = T2 or if variables of type T1 and T2 appear in the same identifier list of a variable, record field or formal parameter list (with the exception of open array types). This kind of type equality is usually called *name equivalence* and is opposed to *structure equivalence*, where two types are "equal" (or at least assignment compatible) if their structures are "equal". Pascal and Modula-2 are representatives for languages using name equivalence, whereas in Modula-3 [Cardelli et al. 1988] structure equivalence prevails.

While name equivalence is the standard equality relationship for types in Oberon, a remarkable exception exists for procedure types. Since the type of a procedure is constituted by its parameter list, i.e. the kind, number and types of the parameters as well as the procedure's result type (if any), and since the type of a procedure is not declared in an explicit type declaration but is specified by the procedure declaration, two procedures (not procedure variables) always have different types. Thus, in order to check whether a procedure can be assigned to a variable of procedure type, the structure of both types, the variable and the procedure type must be compared; i.e. structure equivalence is used. In some cases, this irregularity leads to compiler error messages which are hard to understand by a novice user not familiar with the details. The following Oberon example illustrates some of the problems:

```
TYPE
   T = PROCEDURE (x, y: INTEGER; VAR z: REAL);

VAR
   v: T;

PROCEDURE P (x, y: INTEGER; VAR z: REAL);
PROCEDURE P1 (p: PROCEDURE (x, y: INTEGER; VAR z: REAL));
PROCEDURE P2 (p: T);
```

Due to structure equivalence, it is possible to assign the procedure P to the variable v or to call P1 or P2 with P as actual parameter. However, within P1 it is not allowed to assign the value parameter p to the variable v, since in case of a variable assignment name equivalence is used. Nevertheless, it is possible to call P1 recursively with p (but not with v) as actual parameter, since in this case formal and actual parameter types are identical ("by accident"). On the other hand, within P2 it is allowed to assign p to the variable v and to call P2 recursively using p (or v) as actual parameter but it is not allowed to call P1 with p as parameter.

In Oberon-V structure equivalence is used whenever procedure types are involved; i.e. two procedure types are considered to be "equal", if corresponding parameters and the result types (if any) are equal (see Appendix A.6.5). Thus, the problems mentioned before completely disappear.

2.3.4 Structure Assignment

In contrast to Oberon, in Oberon-V no structure assignment is allowed (with the exception of strings, see Section 2.3.5). Its omission has several reasons:

In numerical programs, structured variables typically are arrays. Copying of arrays should be avoided, since it is a relatively costly operation. Array parameters are usually passed by reference. If it is still inevitable to copy an array, the ALL statement is an appropriate and highly efficient alternative.

If a record type R is exported, not all of its fields have to be exported. These *private fields* are then not visible outside the declaring module. Within a client module, the meaning of an assignment of two variables of type R is difficult to understand, if not all record fields are known. If a private field is a pointer, a record assignment may not be useful at all, or even worse, it may destroy important properties of the exported data structure. Thus, if it is necessary to assign variables of an exported record type, an appropriate copy procedure is probably the better solution.

Structure assignment for variables would consequently demand structure assignment for parameters. Consequently, it should also be possible to specify open array value parameters. While in Oberon the implementation of open array value parameters is difficult but feasible, a similar implementation for Oberon-V would be much more complicated, since then it should also be possible to use an array value constructor as argument (i.e. as actual parameter).

The occurrence of structure assignments within larger programs, e.g. a compiler, may serve as an additional measure for its importance. In the Oberon-V compiler, structure assignments are rarely required, and their replacement by field-wise assignments or call of a copy procedure would impose neither problems nor a loss of efficiency. Last but not least, it is believed that in language design it is a good idea to choose (textually) "small" and unobtrusive symbols to denote efficient or simple operations, whereas costly or difficult operations should be expressed by (textually) "big" and striking language constructs. While this rule is certainly met with the symbol ":=" for the assignment of unstructured types, it is not always for the assignment of structured types.

2.3.5 Character Sequences: The Exception

Strings and character arrays, i.e. character sequences of arbitrary length, play an important role in many programs. Reading of input (scanning) as well as producing output can often be formulated much more elegantly if a language provides an appropriate set of operations defined on strings. While in Pascal [Wirth 1971] and Modula-2 [Wirth 1985a] "string handling" is rudimentarily supported only (it is possible to assign a string to a character array), Oberon allows for direct comparison of strings and character arrays. Furthermore, the standard function COPY relieves the assignment of incompatible character arrays containing character sequences.

In Oberon-V character sequences are almost always treated *specially*: character arrays may be assigned, strings and character arrays may be directly compared as in Oberon and strings can be used as arguments for variable parameters (see below). Within assignments of character arrays, the right-hand side must be a string or a character array containing a character sequence which is copied to the left-hand character array of the assignment. If necessary, the character sequence to be copied is appropriately shortened (i.e. assignment of character arrays has always the semantics of the predefined Oberon procedure COPY). Since no structure assignment is allowed in Oberon-V, the special assignment rules for character arrays do not conflict with other rules for structured assignments.

Because parameters of a structured type must be variable parameters in Oberon-V, the assignment compatibility of string constants and character arrays cannot be used as mechanism to pass string constants as arguments to a procedure. Again, it is *exceptionally* allowed to pass a string constant as argument (i.e. as actual parameter) for a variable parameter. Since variable parameters usually expect variables as arguments, the string constant is first copied into an anonymous variable which is then passed instead.

2.3.6 LOOP Statements Considered Harmful

Roughly speaking, the semantics of statements can be mathematically defined by means of *axioms* which specify *assertions* that must hold *before* and *after* the execution of a particular statement (for an introduction into this topic the reader is referred to the literature, e.g. [Hoare 1969], [Dijkstra 1976] and [Gries 1981]). The essential property of structured statements (i.e. statements which are itself composed of other statements) is that their semantics can be defined "in terms of rules of interference permitting deduction of the properties of the structured statement from the properties of its constituents" [Hoare 1973].

While such an axiomatic specification is easily possible for most structured statements of Oberon [Reiser 1992], it is *not at all* for the *LOOP statement*, since its structure is not completely defined by itself, but depends essentially on the presence or absence and the position of corresponding *EXIT statements* within the loop. From this viewpoint, the LOOP statement cannot honestly be called "well-structured".

Within a compiler, the implementation effort for LOOP statements is higher than for other control structures (except the CASE statement), due to the necessity to handle corresponding EXIT statements in any nesting level. Furthermore, it has been shown that the presence of ill-structured statements such as LOOP and EXIT statements in a programming language significantly complicate the implementation of optimizing compilers [Brandis 1993].

Due to its unimportance for numerical programs and because of the considerations mentioned before, the LOOP statement has been omitted in Oberon-V. It might be noteworthy that its replacement by WHILE loops in the sources of the Oberon-V compiler led to a more readable and therefore more easily understandable program in most cases.

2.3.7 Side-Effect Free Functions

In Oberon, function procedures are ordinary procedures which return a result value; a function procedure must be called within an expression and then stands for its result. The same function procedure may be called several times within an expression, possibly with the same arguments. It is usually expected to return a value dependent on its arguments (and its environment) only. Then, in case of multiple calls of the same function procedure with the same arguments the same result value is returned.

If a function procedure produces *side effects*, i.e. "anything a (function) procedure has done that persists after it returns its value" [Winston 1984], the value of the function may not only depend on its arguments but also on its "calling position", since as a side effect, the procedure may change its environment (through global variables or variable parameters). Function procedures producing side effects are hard to understand and their inappropriate use can lead to unexpected results. The location of bugs due to side effects is often quite difficult and time-consuming. In general, the use of such procedures must be considered as bad programming practice.

However – like all bad habits – as long as such side effects are not explicitly forbidden, they are used. In Oberon-V, functions are not allowed to produce side effects. This is achieved by restricting assignments to local variables only and by prohibiting to call proper procedures within functions. Note that variables referred to by pointers and variable parameters are considered *non-local variables* in Oberon-V. Thus, by simply inspecting the designators on the left-hand side of assignments a compiler may check, whether the restrictions are observed or not.

A similar solution may be found in the programming language Euclid [Lampson et al. 1977], a descendant of Pascal intentionally designed for "the expression of system programs which are to be verified". In Euclid, functions must not have variable parameters nor may a value be assigned to non-local variables. Since non-local (variable) identifiers to be visible within a function or procedure must be "imported" explicitly in Euclid, this import is simply forbidden within functions.

In the Oberon System [Wirth 1988b], almost all side-effect free function procedures obey the restrictions of Oberon-V. Only a few function procedures produce side effects (even visible ones – e.g. a viewer is opened), and they would probably better be written as proper procedures.

3 Implementation

Possible implementation schemes for ALL statements and array constructors are illustrated. The first two sections comprise preliminaries that are required for the understanding of the rest of this chapter. Theorem 3.1 is a customized version of a more general one to be found in [Banerjee 1988] (Theorem 4.2.3, p. 52 ff.). Variable names printed in bold face (e.g. $\mathbf{x}$) are used to denote n-tupels or vectors; i.e. $\mathbf{x} \equiv (x_1, x_2, \dots x_n)$. Operations on vectors are applied element-wise; i.e. $\mathbf{x} + \alpha \equiv (x_1 + \alpha, x_2 + \alpha, \dots x_n + \alpha)$, $\mathbf{x}\alpha \equiv (x_1\alpha, x_2\alpha, \dots x_n\alpha)$ and $\mathbf{x} + \mathbf{y} \equiv (x_1 + y_1, x_2 + y_2, \dots x_n + y_n)$, but $\mathbf{x} \cdot \mathbf{y} \equiv \Sigma\, x_k y_k$ $(1 \le k \le n)$ where $\cdot$ denotes the dot product.

3.1 Linear Functions

A *1-dimensional linear function* f(x) over the integers is a *mapping* of an integer x to an integer y such that $y = cx$, where the *coefficient* c is an integer, too. The set of integers dom(f) over which the function is defined, is called the *domain* of f, and $\mathrm{ran}(f) \equiv \{f(x) : x \in \mathrm{dom}(f)\}$ is called its *range*. The set of these functions is denoted by $LF^1 \equiv \{f : f(x) = cx\}$. A function f is called *total*, if dom(f) covers all integers; and it is called *partial*, if it is only defined over a restricted set. In the following special partial linear functions are considered, with their domains restricted to finite intervals of integers. To be precise, $\mathrm{dom}(f) = \{x : 0 \le x < l\}$ and $l > 0$. The set of these partial functions is denoted by $LF^1(l)$ where l is called the *length* of f, len(f). Thus, such a function f is clearly defined by two integers, namely its length l and the coefficient c, and therefore sometimes $f = LF^1(l, c)$ is written for the specific function $f(x) = cx$ $(0 \le x < l)$.

An *n-dimensional linear function* $f(x_1, x_2, \ldots x_n)$ is a mapping of n integers $x_1, x_2, \ldots x_n$ to an integer y such that $y = c_1x_1 + \ldots c_nx_n$, where all the coefficients c_k ($1 \leq k \leq n$) are integers. The set of all these functions is called $LF^n \equiv \{f : f(\mathbf{x}) = \mathbf{c}\cdot\mathbf{x}\}$. The partial functions discussed in the following are restricted to *rectangular domains* of the form $dom(f) = \{(x_1, x_2, \ldots x_n) : 0 \leq x_1 < l_1, 0 \leq x_2 < l_2, \ldots 0 \leq x_n < l_n\}$ or $\{\mathbf{x} : 0 \leq \mathbf{x} < \mathbf{l}\}$ for short. The set of these partial functions is denoted by $LF^n(\mathbf{l})$ where $\mathbf{l}$ is called the *length* of f, **len**(f). Thus, such a function is clearly defined by 2n integers, namely the 2n components of $\mathbf{l}$ and $\mathbf{c}$ and sometimes $f = LF^n(\mathbf{l}, \mathbf{c})$ is written for a specific function f. Linear functions may be "scaled" and "added"; i.e. the following properties hold:

$$LF^n(\mathbf{l}, \mathbf{c})\alpha \equiv LF^n(\mathbf{l}, \mathbf{c}\alpha)$$

and $$LF^n(\mathbf{l}, \mathbf{c}) + LF^n(\mathbf{l}, \mathbf{d})\ ) \equiv LF^n(\mathbf{l}, \mathbf{c} + \mathbf{d})$$

Sometimes it is useful to know the *minimum* and *maximum* values assumed by partial linear functions. The positive part c^+ and the negative part c^- of an integer c are defined as follows:

$$c^+ \equiv \max(c, 0)$$
$$c^- \equiv \max(-c, 0)$$

Let f be a partial 1-dimensional linear function with len(f) = m+1, i.e. $f = LF^1(m+1, c)$:

thus	$0 \leq x \leq m$
since $c^+ \geq 0$:	$0 \leq c^+x \leq c^+m$
and $-c^- \leq 0$:	$-c^-m \leq -c^-x \leq 0$

After adding the last two sets of inequalities one obtains

	$-c^-m \leq (c^+ - c^-)x \leq c^+m$	
$c^+ - c^- = c$:	$-c^-m \leq cx \leq c^+m$	(∗)

i.e. ran(f) is included in the set $\{-c^-m, -c^-m + 1, \ldots c^+m\}$. Indeed, $-c^-m$ and c^+m are actually values of the function at the end points $x = 0$ and $x = m$ of dom(f):

if $c \geq 0$:	$f(0) = 0 = -c^-m$	$f(m) = cm = c^+m$
if $c \leq 0$:	$f(m) = cm = -c^-m$	$f(0) = 0 = c^+m$

Therefore, $\min(f) = -c^{-}m$ is the minimum value and $\max(f) = c^{+}m$ is the maximum value assumed by the function $f = LF^{1}(m+1, c)$.

Theorem 3.1. Let $f(\mathbf{x})$ be a partial n-dimensional linear function with length $\mathbf{l} = \mathbf{m}+1$, i.e. $f = LF^{n}(\mathbf{m}+1, \mathbf{c})$. Then

$$\min(f) = -\mathbf{c}^{-}\cdot\mathbf{m}$$

and $$\max(f) = \mathbf{c}^{+}\cdot\mathbf{m}$$

Proof. Because of (⁎), for each k, $1 \le k \le n$:

$$-c_k^{-}m_k \le c_k x_k \le c_k^{+}m_k$$

is true and after summation over all k, $1 \le k \le n$:

$$\Sigma -c_k^{-}m_k \le \Sigma c_k x_k \le \Sigma c_k^{+}m_k$$

for short: $$-\mathbf{c}^{-}\cdot\mathbf{m} \le \mathbf{c}\cdot\mathbf{x} \le \mathbf{c}^{+}\cdot\mathbf{m}$$

Again, these values are indeed assumed by $f(\mathbf{x})$, since $f(\mathbf{a}) = -\mathbf{c}^{-}\cdot\mathbf{m}$ and $f(\mathbf{b}) = \mathbf{c}^{+}\cdot\mathbf{m}$ with **a** and **b** defined as follows:

$c_k \ge 0$:	$a_k = 0$	$b_k = m_k$	$(1 \le k \le n)$
$c_k \le 0$:	$a_k = m_k$	$b_k = 0$	$(1 \le k \le n)$

∎

3.2 Affine Functions

An *n-dimensional affine function* f over the integers is a mapping of n integers $\mathbf{x}$ to an integer y such that $y = \mathbf{c}\cdot\mathbf{x} + c_0$. The set of total affine functions is called AF^{n}, whereas $AF^{n}(\mathbf{l})$ denotes the set of affine functions restricted to a rectangular area with $\mathbf{l} = \mathbf{len}(f)$. A specific affine function f is written as $f = AF^{n}(\mathbf{l}, \mathbf{c}, c_0)$; f is clearly defined by 2n+1 values. Besides "scaling" and "adding", affine functions may be "moved" and the resulting functions are still affine functions; i.e. the following identities hold:

$$AF^{n}(\mathbf{l}, \mathbf{c}, c_0) + \alpha \equiv AF^{n}(\mathbf{l}, \mathbf{c}, c_0 + \alpha)$$

and $$AF^{n}(\mathbf{l}, \mathbf{c}, c_0)\alpha \equiv AF^{n}(\mathbf{l}, \mathbf{c}\alpha, c_0\alpha)$$

and $$AF^{n}(\mathbf{l}, \mathbf{c}, c_0) + AF^{n}(\mathbf{l}, \mathbf{d}, d_0) \equiv AF^{n}(\mathbf{l}, \mathbf{c} + \mathbf{d}, c_0 + d_0)$$

Linear functions may be regarded as a special case; i.e. $LF^n(\mathbf{l}, \mathbf{c}) = AF^n(\mathbf{l}, \mathbf{c}, 0)$ and one may write $LF^n(\mathbf{l}, \mathbf{c}) = AF^n(\mathbf{l}, \mathbf{c}, c_0) - c_0$. Using this relationship, Theorem 3.1 is easily adapted to affine functions and then leads to

Theorem 3.2. Let $f = AF^n(\mathbf{m}+1, \mathbf{c}, c_0)$. Then $\min(f) = c_0 - \mathbf{c}^{-}\cdot\mathbf{m}$, and $\max(f) = c_0 + \mathbf{c}^{+}\cdot\mathbf{m}$.

3.3 Representation of Expressions by Expression Trees

Expressions consist of *operands* and *operators*; operands are either literals (i.e. constant values), designators or expressions themselves. Operators combine one or several operands with an operation. Note: in Oberon-V there are also sets and functions calls. Sets may be reduced to expressions consisting of literals and designators, whereas function calls may be thought of being treated as special (user-defined) operators.

Expressions can be represented by expression trees. Their nodes are either *leaf nodes* representing literals or designators, or *inner nodes* representing operations (Figure 3.1). Inner nodes usually have one or two successors: their operands. General techniques for the construction of expression trees from textual representations of expressions are well-known and not explained further (cf. [Aho 1977]).

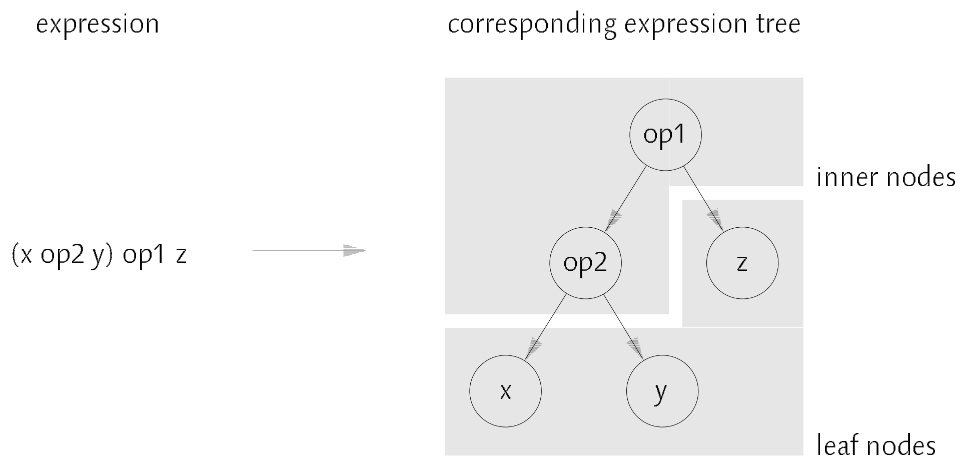


Figure 3.1 Expression Trees

A designator consists of a (variable) identifier followed by selectors which again may contain expressions, e.g. an expression i denoting the element index within a designator of the form A[i]. Variable access can be regarded as *dereferencing* the variable's *address*. If a variable's address is not only allowed to be a *constant address* but also an *address expression*, all forms of selectors can be represented in a unified manner by using a special *VAR* node denoting *dereferenciation* of an address expression.

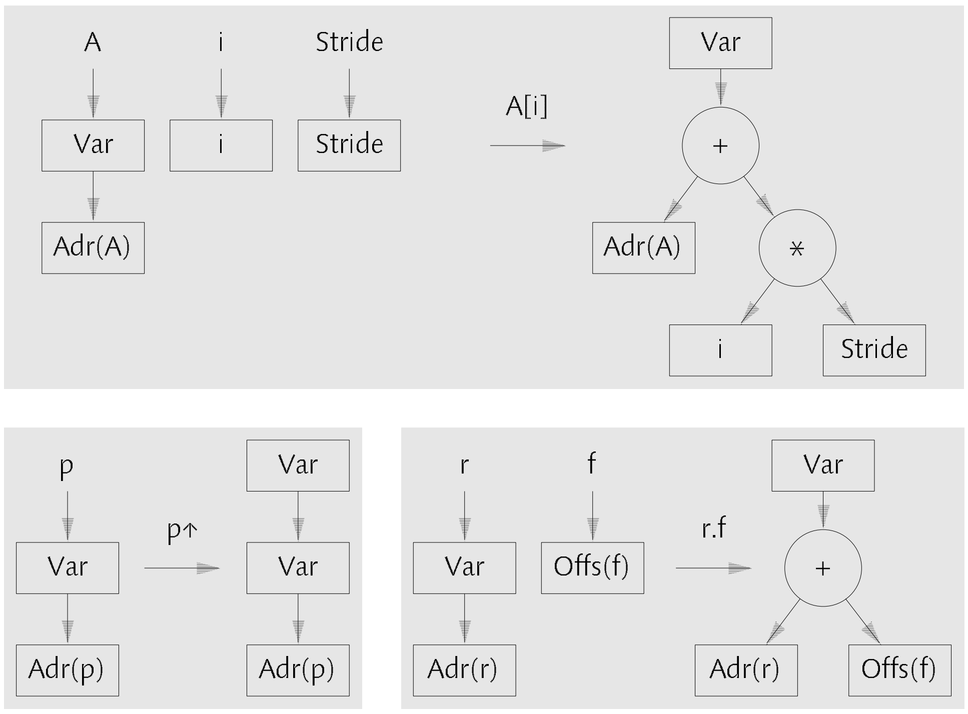


Figure 3.2 Unified Representation of Selectors

Figure 3.2 depicts the construction of expression trees from the component expressions. It shows the trees for array indexing, pointer dereferenciation and record field selection. Thus, all *address computations* are explicitly visible in the expression tree. The advantage of such a representation is the possibility to restructure not only expressions but also address computations, e.g. for optimization purposes. Note: often address computations are not explicitly visible in expression trees, but special nodes are used to specify different selectors (e.g. in the front end of the portable Oberon compiler OP2 [Crelier 1990]).

A complete example of an assignment statement and its corresponding expression tree using the unified representation of selectors may be found in Figure 3.3. Obviously, not only expressions but entire statements and programs may be represented in form of trees which then are called *syntax trees*.

```
CONST
  N = 100;

VAR
  A: ARRAY N OF REAL;
  t, x, y: REAL;
  i: INTEGER;

BEGIN
    ...
    t := A[i]⁎x + y
```

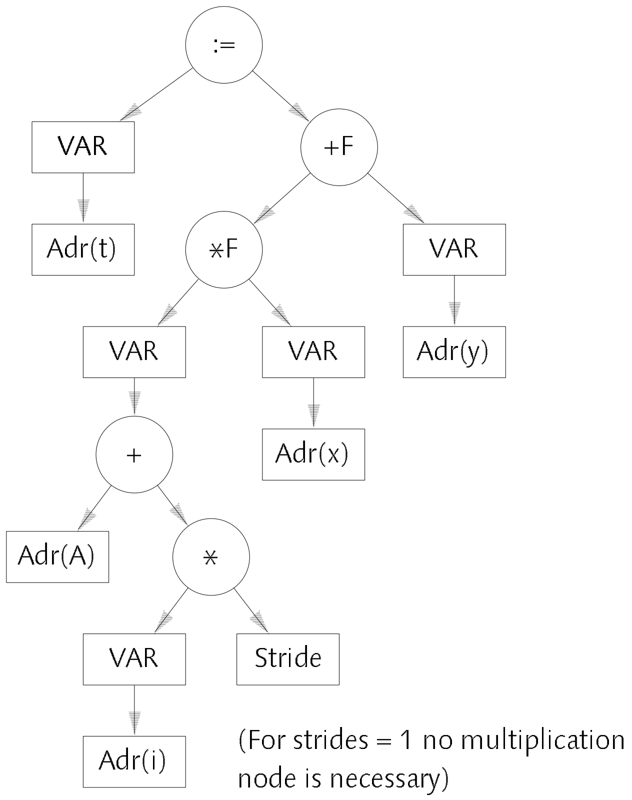


Figure 3.3 Syntax Tree of an Assignment

3.4 Compilation of Range Declarations

Range declarations precede assignment sequences in ALL statements as well as expressions in array constructors. A 1-dimensional range declaration introduces a single range identifier r and associates it with a range. In the following it will be shown that every range identifier denotes the range ran(f) of a *partial affine function* f; and vice versa that ran(f) of every partial affine function f may be expressed by a *range expression*, i.e. an expression containing range identifiers as operands. This correspondence will be used to represent a range within a compiler's data structures and also explains the name *range identifier*.

Let r be a range identifier associated with the range a .. b; i.e. the range standing for the set of integers {a, a+1, ... b}. Then, for a function f = AF^1(b–a+1, 1, a) the range is ran(f) = {1x + a : 0 ≤ x < b–a+1} = {a, a+1, ... b}, i.e. the range of the function corresponds to the range a .. b and the length of the function corresponds to the length of the range. Vice versa, let f be an affine function with f = AF^1(l, c, c_0). Then, the corresponding range ran(f) can be denoted by the range expression r⁎c + c_0, where r is a range identifier declared as r = 0 .. l–1.

Thus, a range r – which is also a range expression – can be *identified* with the corresponding partial affine function f. Since a specific function $f = AF^1(l, c, c_0)$ is clearly determined by 3 values, a range can be represented by these values.

In general, a range declaration introduces more then one range identifier. As will be seen in Section 3.5, computations are simplified if the correspondence between a *single* range r and a *1*-dimensional affine function f is generalized to *n*-dimensional range declarations and functions. Let us consider the following n-dimensional range declaration:

$$r_1 = a_1 \,..\, b_1,\ r_2 = a_2 \,..\, b_2,\ \ldots\ r_n = a_n \,..\, b_n$$

Each range r_i $(1 \le i \le n)$ corresponds to a function $f_i \in AF^n$ by the following relationship (with δ_{ik} denoting the Kronecker symbol, i.e. $\delta_{ik} = 1$ for $i = k$ and $\delta_{ik} = 0$ for $i \ne k$):

$$r_i = a_i \,..\, b_i \quad \leftrightarrow \quad f_i = AF^n(\mathbf{l}, \mathbf{c}_i, c_{i0}) \text{ with } c_{ik} = \delta_{ik} \text{ and } c_{i0} = a_i \quad (1 \le i, k \le n)$$

and the length $\mathbf{l}$ is the same for all functions f_i; namely $l_i = b_i - a_i + 1$ $(1 \le i \le n)$; i.e. each range r_i is identified with an n-dimensional partial affine function which "ignores" the values of all x_k with $i \ne k$ and "behaves" like in the 1-dimensional case for x_i. Since the length $\mathbf{l}$ is the same for all functions f_i, it suffices to store it only once. Each range of an n-dimensional range declaration is clearly defined by n+1 values (the n values of $\mathbf{c}_i$ plus c_{i0}). The following example illustrates the situation: the 3-dimensional range declaration

```
CONST
   n = 10;
VAR
   a, b: INTEGER;
...
u = 0 .. n–1, v = –10 .. 10, w = a .. b
```

declares 3 range identifiers with the associated affine functions and their coefficients:

Function	*Coefficients* $(c_{i0}, c_{i1}, c_{i2}, c_{i3})$
$f_u(\mathbf{x}) = (1, 0, 0)\cdot\mathbf{x} + 0$	0, 1, 0, 0
$f_v(\mathbf{x}) = (0, 1, 0)\cdot\mathbf{x} + (-10)$	–10, 0, 1, 0
$f_w(\mathbf{x}) = (0, 0, 1)\cdot\mathbf{x} + a$	a, 0, 0, 1

In addition, there is the length **l** = (10, 21, b–a+1) which is simply associated with the entire range declaration (thus, it makes sense to speak of the *length* of an n-dimensional range declaration). During compilation a *range object* is associated to each range identifier. The range object simply contains the coefficients of the corresponding affine function. Note that a range specification may contain variables and entire expressions; i.e. the coefficients must be held in a suitable form, e.g. an array of expression trees (cf. Section 5.2).

Altogether, a range declaration is compiled as follows: first of all, a new identifier scope has to be opened (by an appropriate procedure of the compiler's symbol table manager), since range identifiers are local to the enclosing ALL statement or array constructor. Each new range identifier is then inserted in the top scope of the symbol table (and an error is raised, if the range identifier has been declared already in the same scope). For each range a .. b, a range object is created which represents the function denoted by the range (i.e. which holds the n+1 coefficients), and the length corresponding to the entire range declaration is updated. The range object is then bound to its associated range identifier(s). After the range declaration has been processed, a range object referred to by its associated identifier can be found with a simple table lookup. At the end of the enclosing ALL statement or array constructor the scope is closed again. Figure 3.4 shows the contents of the top scope of a compiler's symbol table after processing of the previous range declaration example.

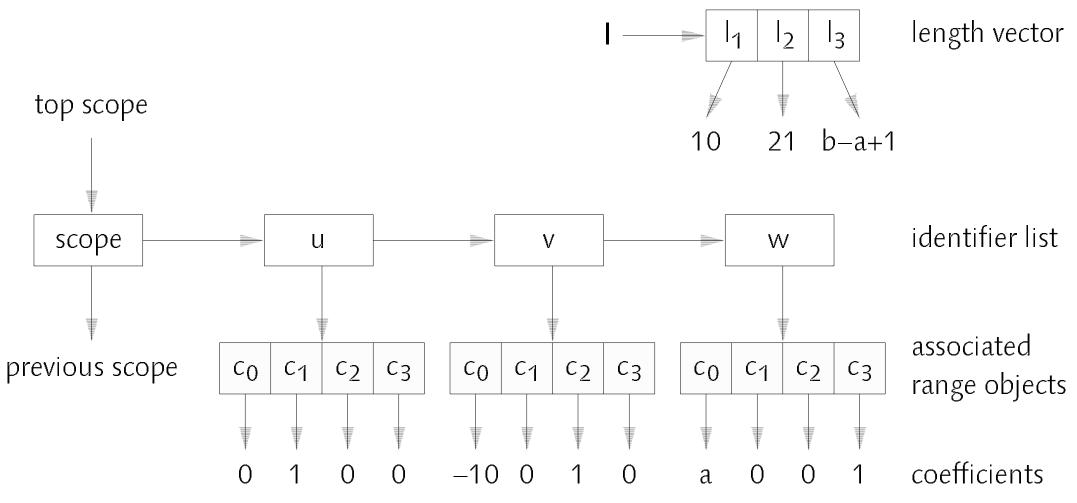


Figure 3.4 Symbol Table Contents after Processing a Range Declaration

Note that this compilation scheme automatically leads to a *canonical representation of range declarations,* since every n-dimensional range declaration is reduced to a length vector **l** associated with the entire declaration, and a list of n range objects which may be referred to by their associated range identifiers and which are represented by the coefficients of their corresponding affine functions. Thus, instead of an n-dimensional range declaration of the form

$$r_1 = a_1 \,..\, b_1,\ r_2 = a_2 \,..\, b_2,\ \ldots\ r_n = a_n \,..\, b_n$$

also the *canonical range declaration*

$$r_1' = 0 \,..\, l_1 - 1,\ r_2' = 0 \,..\, l_2 - 1,\ \ldots\ r_n' = 0 \,..\, l_n - 1$$

with $r_k = r_k' + a_k$ and $l_k = b_k - a_k + 1$ ($1 \le k \le n$) may be used instead without loss of generality. In this case $\mathbf{l} = (l_1, l_2, \ldots l_n)$ denotes the *length of the range declaration*.

3.5 Linear Range Expressions

Range expressions are expressions which have ranges (denoted by range identifiers) as operands. Each range expression must be considered within its environment, i.e. within the scope of the corresponding n-dimensional range declaration. Within a particular range expression, only the range identifiers introduced by this declaration can occur.

A conventional expression may be represented by an expression tree (cf. Section 3.3) and, of course, this holds for range expressions too. Within an expression tree, a range is represented by its associated range object which has been created during the range declaration. During parsing of an expression, the expression tree grows. Each operator applied to its operands usually leads to a new tree node denoting the operation and pointing to its operand trees. Range expressions are treated in a special way when *linear operations* are involved (+, – and ∗ with a scalar value): since each range denotes a function $f \in AF^n(\mathbf{l})$ and since this function class is closed under addition and multiplication with a scalar value, this holds for ranges too. In the example

```
VAR
   a, b: INTEGER;
...
ALL r = a .. b DO
   ... r∗10 + 4 ...
```

the range identifier r denotes a range object corresponding to the function $f = AF^1(b-a+1, 1, a)$. Therefore the expression r∗10 + 4 corresponds to the *function* $f*10 + 4 = AF^1(b-a+1, 10, a*10 + 4)$ (cf. Section 3.2). Thus, *linear expressions* over ranges can be transformed into linear expressions over the ranges' coefficients. Figure 3.5 illustrates this transformation for the example. The lengths of the ranges are never changed (they are not shown in the figure). Note that for each transformation a new range object must be generated (r' and r'' in Figure 3.5), because a range object may be shared by different expression trees.

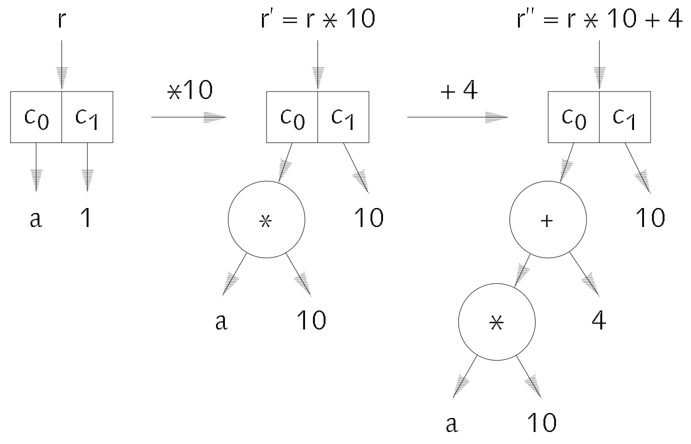


Figure 3.5 Transformation of Range Expressions

According to the rules for affine functions (Section 3.2), the same transformations can be performed for $n > 1$. The range objects declared by an n-dimensional range declaration correspond to functions $f \in AF^n$. Since all lengths and the number of dimensions of all these functions are the same within the scope of the range declaration, all linear operations are well-defined. Within the scope of a range declaration, the objects involved in range expressions correspond either to scalar values or such functions f. Thus, if the expression is *linear*, the entire expression can be represented by a single range object (cf. also Section 3.8.1). Especially, also expressions of the form $a_0 + r_1*s_1 + r_2*s_2 + ... + r_n*s_n$, where a_0 and the s_k's designate scalar values and the r_k's designate range identifiers ($1 \le k \le n$), can be represented by a single range object. Such computations are caused implicitly when subscripting an n-dimensional array A with n ranges r_k. In this case, a_0 denotes the address of A and the s_k's denote the strides.

3.6 Reorganization Properties of ALL Statements

An ALL statement consists of an n-dimensional range declaration followed by a sequence of assignments. In this section it will be shown that this assignment sequence can be reorganized in such a way that the overall semantics of the

ALL statement is not changed, but a potentially parallel execution becomes possible. A general ALL statement containing m assignments A_k is of the form

```
ALL $r_1 = a_1 .. b_1, r_2 = a_2 .. b_2, ... r_n = a_n .. b_n$ DO
   $A_1(r_1, r_2, ... r_n)$;
   $A_2(r_1, r_2, ... r_n)$;
   ...
   $A_m(r_1, r_2, ... r_n)$
END
```

where each $A_k(r_1, r_2, ... r_n)$ may contain the range identifiers $r_1, r_2, ... r_n$. Instead of this ALL statement, its *canonical form* may be used without loss of generality, i.e. the range declaration may be replaced by its canonical form (cf. Section 3.4). Then, the general ALL statement looks like

```
ALL $\mathbf{r} = 0 .. \mathbf{l}-1$ DO
   $A_1(\mathbf{r})$;
   $A_2(\mathbf{r})$;
   ...
   $A_m(\mathbf{r})$
END
```

using an informal notation where $\mathbf{l}$ is the length of the range-declaration (cf. Section 3.4). According to the definition of the ALL statement, a legal interpretation of it is (using a pseudo code notation):

```
FOREACH $\mathbf{i}$ IN $\{\mathbf{x} : 0 \le \mathbf{x} < \mathbf{l}\}$ DO
   $A_1(\mathbf{i})$;
   $A_2(\mathbf{i})$;
   ...
   $A_m(\mathbf{i})$
END
```

After numbering the elements of the set $\mathbf{R} = \{\mathbf{x} : 0 \le \mathbf{x} < \mathbf{l}\}$, an explicit sequence of m∗v assignments may be written instead, with $v = l_1 * l_2 * ... * l_n$ (i.e. v is the number of elements in $\mathbf{R}$):

```
$A_1(\mathbf{i}_1)$; $A_2(\mathbf{i}_1)$; ... $A_m(\mathbf{i}_1)$;          (* $S(\mathbf{i}_1)$ *)
$A_1(\mathbf{i}_2)$; $A_2(\mathbf{i}_2)$; ... $A_m(\mathbf{i}_2)$;          (* $S(\mathbf{i}_2)$ *)
...                                      ...
$A_1(\mathbf{i}_v)$; $A_2(\mathbf{i}_v)$; ... $A_m(\mathbf{i}_v)$           (* $S(\mathbf{i}_v)$ *)
```

where each $\mathbf{i}_k$ ($1 \leq k \leq v$) stands for a different element of **R**. The definition of ALL statements requires that any two assignment sequences $S(\mathbf{i}_k)$ and $S(\mathbf{i}_l)$ with $1 \leq k, l \leq v$ and $k \neq l$ be *independent* of each other; i.e. no variables that have been accessed or modified by $S(\mathbf{i}_k)$ are modified by $S(\mathbf{i}_l)$ and vice versa. Hence, it is allowed to reorder the assignment sequence to the following form without changing its effect:

$A_1(\mathbf{i}_1)$; $A_1(\mathbf{i}_2)$; ... $A_1(\mathbf{i}_v)$;
$A_2(\mathbf{i}_1)$; $A_2(\mathbf{i}_2)$; ... $A_2(\mathbf{i}_v)$;
...
$A_m(\mathbf{i}_1)$; $A_m(\mathbf{i}_2)$; ... $A_m(\mathbf{i}_v)$

This assignment sequence is now considered at a finer level of granularity: each assignment $A(\mathbf{i})$ is of the form $D(\mathbf{i}) := E(\mathbf{i})$ where D stands for a designator and E stands for an expression; i.e. the assignment sequence can be written in the form

$D_1(\mathbf{i}_1) := E_1(\mathbf{i}_1)$; $D_1(\mathbf{i}_2) := E_1(\mathbf{i}_2)$; ... $D_1(\mathbf{i}_v) := E_1(\mathbf{i}_v)$;
$D_2(\mathbf{i}_1) := E_2(\mathbf{i}_1)$; $D_2(\mathbf{i}_2) := E_2(\mathbf{i}_2)$; ... $D_2(\mathbf{i}_v) := E_2(\mathbf{i}_v)$;
...
$D_m(\mathbf{i}_1) := E_m(\mathbf{i}_1)$; $D_m(\mathbf{i}_2) := E_m(\mathbf{i}_2)$; ... $D_m(\mathbf{i}_v) := E_m(\mathbf{i}_v)$

Because functions and hence expressions are *side-effect free*, and due to the *independence* of different assignment sequences $S(\mathbf{i}_k)$ and $S(\mathbf{i}_l)$, it is even possible to use *parallel assignments* instead. For parallel assignments, firstly all expressions on the right-hand side of the assignment are evaluated and then the results are simultaneously assigned to the corresponding variables on the left-hand side:

$D_1(\mathbf{i}_1), D_1(\mathbf{i}_2), \ldots D_1(\mathbf{i}_v) := E_1(\mathbf{i}_1), E_1(\mathbf{i}_2), \ldots E_1(\mathbf{i}_v)$;
$D_2(\mathbf{i}_1), D_2(\mathbf{i}_2), \ldots D_2(\mathbf{i}_v) := E_2(\mathbf{i}_1), E_2(\mathbf{i}_2), \ldots E_2(\mathbf{i}_v)$;
...
$D_m(\mathbf{i}_1), D_m(\mathbf{i}_2), \ldots D_m(\mathbf{i}_v) := E_m(\mathbf{i}_1), E_m(\mathbf{i}_2), \ldots E_m(\mathbf{i}_v)$

Note that this sequence of m parallel assignments still has the same effect as the original ALL statement because only semantics-preserving transformations have been applied. On a vector computer, the expressions $E_k(\mathbf{i}_1)$, $E_k(\mathbf{i}_2)$, ... $E_k(\mathbf{i}_v)$ ($1 \leq k \leq m$) may be implemented as vector expressions on vectors of length v. Since v is relatively large in general, it may be necessary to divide the vectors into a sequence of shorter vectors or *vector slices* (cf. Section 1.2). The reader may see himself that such a division still preserves the semantics of the ALL statement.

Altogether one may conclude that a sequence of independent scalar assignments enclosed in an n-dimensional ALL statement of the form

```
ALL $\mathbf{r}$ = 0 .. $\mathbf{l}$–1 DO
  $A_1(\mathbf{r})$;
  $A_2(\mathbf{r})$;
  ...
  $A_m(\mathbf{r})$
END
```

can be translated into a *sequence of vector assignments* of the form

```
$A_1(\mathbf{v})$;
$A_2(\mathbf{v})$;
...
$A_m(\mathbf{v})$
```

where $\mathbf{v}$ denotes the vector of the elements of $\mathbf{R}$ (i.e. $\mathbf{v} = (\mathbf{i}_1, \mathbf{i}_2, \dots \mathbf{i}_v)$ and $\mathbf{i}_k \in \mathbf{R}$ for $1 \le k \le v$), without changing its original effect.

3.7 Translation of ALL Statements into Vector Instructions

With the prerequisites leading the way to this section, half the work necessary for the translation of ALL statements is already explained. After compilation of the range declaration as described in Section 3.4, the assignment sequence has to be parsed and a list of syntax trees for these assignments is built (cf. Section 3.3). Due to the properties of the ALL statement (Section 3.6) these syntax trees can be regarded as a sequence of *vector assignments* consisting of a designator denoting a vector on the left-hand side and a vector expression on the right-hand side of each assignment. Section 3.7.1 explains the code generation for 1-dimensional ALL statements, i.e. ALL statements containing a 1-dimensional range declaration. The techniques are generalized to n-dimensional ALL statements in Section 3.7.2. A few interesting cases such as array accesses of the form A[B[i]] and conditional assignments are considered in Section 3.7.3. The handling of index checks is omitted here but explained in Section 3.9.

3.7.1 1-dimensional ALL Statements

In case of a 1-dimensional ALL statement, vector instructions can be generated without further reorganization of the syntax tree: as for scalar expressions, the assignment list is traversed and code is generated by recursively traversing the left-hand and right-hand side of the expression trees for each assignment (code generation for scalar expression trees is not explained here; the reader is referred to standard literature on compiler construction, e.g. [Aho 1977]). In most cases, a variable denoting a vector is represented by a VAR node and its address which is a range object (see Sections 3.3 and 3.5, special situations are described in Section 3.7.3). In the 1-dimensional case, this range object contains two coefficients c_0 and c_1 only representing the *vector address* (Section 1.1) which directly may be used to generate the necessary vector load or store instruction. However, since the length of vectors is usually longer then the maximum vector length VL_{max} of a vector register or even unknown at compile time, the vectors have to be sliced; i.e. a loop enclosing the vector assignments and additional slice pointers are required (cf. Section 1.2). As an example, a simple ALL statement is considered:

```
VAR
  A, B: ARRAY 1000 OF REAL;
  x: REAL;
  e1, e2: INTEGER;
...
ALL i = 0 .. 99 DO
  A[i * e1] := B[i * e2] * x
END
```

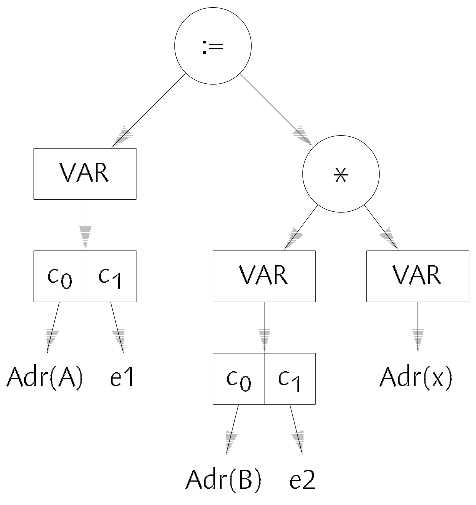


Figure 3.6 ALL Statement and Corresponding Syntax Tree

After processing the ALL statement, the syntax tree illustrated in Figure 3.6 is obtained plus the length $l = 100$ of the range declaration (not shown in the figure). For a real machine with $VL_{max} < l$ (or l unknown at compile time), a slicing loop is necessary and the code may then look as follows (compiled for the hypothetical vector computer introduced in Section 1.1):

```
;          S0                    address of A
;          S1                    address of B
;          S2                    variable x
;          S3                    variable e1
;          S4                    variable e2
;          S5                    counter          (introduced by the compiler)
;          S6                    slice pointer A↑ (introduced by the compiler)
;          S7                    slice pointer B↑ (introduced by the compiler)
;
           S5 := 100             ; initialize counter c with range length
           S6 := S0              ; initialize A↑
           S7 := S1              ; initialize B↑
Loop       S10 := S5 > 0         ; c > 0
           jump Exit (~S10)      ; WHILE c > 0 DO
           VL := S5              ; set vector length
|          V11 := M[S4, S7]      ; B[i ⁎ e2]
|          V12 := S2 ⁎F V11      ; B[i ⁎ e2] ⁎ x
|          M[S3, S6] := V12      ; A[i ⁎ e1] := B[i ⁎ e1] ⁎ x
           S13 := VL             ; get vector length of this iteration
           S14 := S3 ⁎ S13       ; e1 ⁎ VL
           S6 := S6 + S14        ; A↑ := A↑ + e1 ⁎ VL
           S15 := S4 ⁎ S13       ; e2 ⁎ VL
           S7 := S7 + S15        ; B↑ := B↑ +  e2 ⁎ VL
           S5 := S5 – S13        ; c := c – VL
           jump Loop             ; END
Exit       ...
```

The code for the vector expression (marked by a vertical bar on the left margin) corresponds directly to the expression tree in Figure 3.6, with the only exception that the coefficients c_0 have been replaced by auxiliary variables (namely the slice pointers A↑ and B↑). Thus, for a 1-dimensional ALL statement where the expression tree contains m range objects, in general m+1 additional variables have to be introduced by the compiler: m variables for the slice pointers and 1 variable for the loop counter. Furthermore, note that the strides of A and B and the scaling factor x are denoted by simple variables (e1, e2 and x) in this example. In general, strides and scalar values may be arbitrary expressions. Beacuse these scalar expressions do not change their values during execution of an ALL statement, they could be computed once before entering the slicing loop (if these expressions would change their values, different assignment sequences of the ALL statement would not be independent). These optimizations and the introduction of the slice pointers can be obtained by a simple traversal of the syntax tree corresponding to the assignment sequence. During this traversal, the following substitutions are done:

1. Introduce a new (anonymous) variable v for each *scalar expression* E which is not *simple*, i.e. which is not simply a literal or an unstructured variable, or which is the coefficient c_0 of a range object.
2. Substitute this scalar expression E by the newly introduced variable v in the expression tree.
3. Generate an assignment statement "v := E", i.e. the corresponding syntax tree, and append it to a special list L1 of assignments which have to be executed *before* entering the slicing loop of the ALL statement.
4. If the substituted expression E was a c_0-coefficient of a range object, then generate an assignment statement "v := v + c_1*VL" and append it to a special list L2 of assignments which have to be executed *at the end* of each slicing-loop iteration.

After these transformations, a 1-dimensional ALL statement can be translated to machine code in 5 steps (if also run-time index checks are to be generated, an additional step is necessary; cf. Section 3.9):

1. Generate code for the assignment sequence L1.
2. Generate code for the slicing loop header, i.e. introduce a new counter variable c (preferably to be held in a register), initialize it with the length of the range and generate code to check the termination condition.
3. Generate code for the vector assignments by traversing the syntax trees.
4. Generate code for the assignment sequence L2.
5. Generate code for the slicing loop end, i.e. decrement the counter variable c by (the current value of) VL and generate a jump to the beginning of the loop.

In case of a range with constant length not greater then VL_{max}, neither a slicing loop nor additional variables have to be introduced but code can be generated directly by traversing the expression tree. Thus, using a constant length VL_{max}, vector expressions can be mapped one-to-one to the machine's vector instructions.

3.7.2 *n-dimensional ALL Statements*

The basic insight leading to an implementation of multi-dimensional ALL statements is that a n-dimensional ALL statement of the form

```
ALL $\mathbf{r}$ = 0 .. $\mathbf{l}$–1 DO
   S($\mathbf{r}$)
END
```

can be *emulated* using n loops and n additional variables x_1, x_2, ... x_n:

```
$x_1$ := $l_1$;
WHILE $x_1$ > 0 DO
   $x_2$ := $l_2$;
   WHILE $x_2$ > 0 DO
      ...
         $x_n$ := $l_n$;
         WHILE $x_n$ > 0 DO
            VL := ($x_n$ – 1) MOD $VL_{max}$ + 1;
            ALL r' = 0 .. VL–1 DO
               S'(r')
            END;
            $x_n$ := $x_n$ – VL
         END
      ...
      $x_2$ := $x_2$ – 1
   END;
   $x_1$ := $x_1$ – 1
END
```

The innermost loop proceeds in steps of VL with VL $\leq VL_{max}$. Since the enclosed ALL statement processes vectors of length VL, no slicing loop is required and the statement sequence S'(r') can directly be implemented using vector instructions (cf. previous Section). Thus, the key to the implementation of n-dimensional ALL statements is the transformation of the assignment sequence S($\mathbf{r}$) containing n different ranges r_1, r_2, ... r_n to an assignment sequence S'(r') using only a single range identifier r' denoting a range of length VL. In the following this goal is approached in a few small steps.

Within the syntax trees of an n-dimensional ALL statement, a range object represents a function f ∈ $AF^n(\mathbf{l})$. For each element e of the range ran(f) of such a function, the assignment sequence within the ALL statement must be executed once. In case of a 1-dimensional function $f(x) = c_1x + c_0$ with $0 \leq x < l$, the range corresponds directly to the vector of elements (e) = (c_0, $c_0 + c_1$, $c_0 + 2c_1$, ... $c_0 + (l–1)c_1$) and hence vector instructions can be generated immediately

(Section 3.7.1). In case of an n-dimensional function $f(\mathbf{x}) = \mathbf{c} \cdot \mathbf{x} + c_0$, all elements e of ran(f) can be enumerated using n loops and n variables $x_1, x_2, \dots x_n$ as indicated in the beginning:

```
$x_1 := l_1$;
WHILE $x_1 > 0$ DO $x_1 := x_1 - 1$;
  $x_2 := l_2$;
  WHILE $x_2 > 0$ DO $x_2 := x_2 - 1$;
    ...
      $x_n := l_n$;
      WHILE $x_n > 0$ DO $x_n := x_n - 1$;
        $e := f(x_1, x_2, \dots x_n)$
      END
    ...
  END
END
```

Since f is an affine function, the enumerated values e can be computed *incrementally*. In this case, n auxiliary variables v_k are needed and the variables x_k degenerate to counter variables:

```
$v_1 := c_0$; $x_1 := l_1$;
WHILE $x_1 > 0$ DO
  $v_2 := v_1$; $x_2 := l_2$;
  WHILE $x_2 > 0$ DO
    ...
      $v_n := v_{n-1}$; $x_n := l_n$;
      WHILE $x_n > 0$ DO
        $e := v_n$;
        $v_n := v_n + c_n$; $x_n := x_n - 1$
      END
    ...
    $v_2 := v_2 + c_2$; $x_2 := x_2 - 1$
  END;
  $v_1 := v_1 + c_1$; $x_1 := x_1 - 1$
END
```

If the innermost loop proceeds faster than in steps of 1, e.g. in steps of VL, the elements are enumerated in form of entire vectors (e) and then the following code is obtained (only the innermost loop is shown):

```
...
   $v_n := v_{n-1}$; $x_n := l_n$;
   WHILE $x_n > 0$ DO
      VL := ($x_n$ – 1) MOD $VL_{max}$ + 1;
      (e) := ($v_n$, $v_n + c_n$, $v_n + 2c_n$, ... $v_n$ + (VL–1)$c_n$);
      $v_n := v_n + VL*c_n$; $x_n := x_n - VL$
   END
...
```

Thus, the enumeration of ran(f) for f ∊ $AF^n(\mathbf{l})$ has been reduced to the enumeration of ran(f') with $f'(x) = v_n + c_n x$ and f' ∊ AF^1(VL). The same transformations can be applied to *every* range object within an n-dimensional ALL statement. After these transformations, every range object corresponding to a function f ∊ $AF^n(\mathbf{l})$ has been substituted by a 1-dimensional range object corresponding to a function f' ∊ AF^1(VL). Then, for every range object r, n auxiliary variables v_{kr} ($1 \le k \le n$) are required. For a given k, all these v_{kr} must be initialized before entering the loop k and incremented by the corresponding coefficient c_{kr} in the loop k, thus leading to the following loop structure for the ALL statement (remember the special instruction available for the computation of VL; cf. Section 1.1):

```
*initialize all $v_{1r}$*; (* *prolog 1* *)
$x_1 := l_1$;
WHILE $x_1 > 0$ DO
   *initialize all $v_{2r}$*; (* *prolog 2* *)
   $x_2 := l_2$;
   WHILE $x_2 > 0$ DO
      ...
         *initialize all $v_{nr}$*; (* *prolog n* *);
         $x_n := l_n$;
         WHILE $x_n > 0$ DO
            VL := ($x_n$ – 1) MOD $VL_{max}$ + 1;
            ALL r' = 0 .. VL–1 DO
               S'(r')
            END;
            *increment all $v_{nr}$*; (* *epilog n* *)
            $x_n := x_n - VL$
         END
      ...
      *increment all $v_{2r}$*; (* *epilog 2* *)
      $x_2 := x_2 - 1$
   END;
   *increment all $v_{1r}$*; (* *epilog 1* *);
   $x_1 := x_1 - 1$
END
```

The range transformation algorithm is described by the pseudo code procedure *SubstituteRange*. It is assumed that an expression tree is represented by *Expression* nodes pointing to their operands. Among others, operands may be range objects represented by *Range* records or VAR nodes represented by *Variable* records (see below). Note that each coefficient r.c[k] of a range object r is again an expression tree (possibly holding a constant object only). Since VAR nodes and range objects may occur at any position within an expression tree, they must be freely interchangeable. The type definitions below indicate an implementation using *type extension* (for implementation details, cf. Chapter 5). The goal of SubstituteRange is to substitute all range objects with n coefficients by range objects with only two coefficients and thereby to determine the necessary initialization and increment statements for the newly introduced variables v_{kr}. These statements are held in n assignment lists called Prolog[k] and Epilog[k] respectively and subsequently can be used to generate the n loops for the ALL statement.

```
TYPE
   Expression = RECORD ... END;
   Variable = RECORD (Expression) ... END;
   Range = RECORD (Expression) c: ARRAY n+1 OF ↑Expression; ... END;

VAR
   Prolog: ARRAY n+1 OF assignment list;
   Epilog: ARRAY n+1 OF assignments list;

PROCEDURE SubstituteRange (VAR r: ↑Range);
   VAR k: INTEGER; h, v: ↑Variable; pv, c: ↑Expression;
BEGIN
   pv := r.c[0]; k := 1;
   WHILE k <= n DO
      c := r.c[k];
      IF ~Simple(c) THEN
         allocate a new variable h, generate an assignment "h := c" and append it to Prolog[1];
         c := h
      END;
      allocate new variable v;
      generate an assignment "v := pv" and append it to Prolog[k];
      IF k = n THEN generate an assignment "v := v + VL * c" and append it to Epilog[k]
      ELSE generate an assignment "v := v + c" and append it to Epilog[k]
      END;
      pv := v; k := k + 1
   END;
   generate a new range r' with r'.c[0] := pv and r'.c[1] := c;
   r := r'
END SubstituteRange;
```

SubstituteRange processes all coefficients c = r.c[k] of a range r sequentially: a new variable v is introduced (corresponding to the variables v_{kr}) and syntax trees corresponding to the assignment "v := pv" and "v := v + c" are generated and appended to the list of prolog and epilog assignments of the k-th loop. The variable pv is a reference to the previously introduced variable v and corresponds to v_{k-1r}. In case of the innermost loop (k = n), v is incremented by VL∗c since vector instructions are used and hence VL elements are processed in each loop iteration. As an optimization, coefficients c which are not *simple*, i.e. which are not simply constant values or variables with simple address, are substituted by variables h which then are initialized outside the loop nest (i.e. in Prolog[1]).

SubstituteRange is easily extended such that temporary variables are introduced for *all* scalar expression that are not simple (cf. Section 3.7.1). If l_n is known at compile time and $l_n \leq VL_{max}$ for an n-dimensional ALL statement, the innermost loop is not required. Furthermore, if $l_n \leq VL_{max}$ and n = 1, no loop and also no additional variables are required (both optimizations are omitted in SubstituteRange).

In some cases, a translation of the ALL statement into scalar instructions is desired, either because certain vector instructions are missing or because no vector instructions are available at all. Then, SubstituteRange must substitute all range objects by scalar slice pointers. The necessary modifications at the end of the procedure SubstituteRange are shown below:

```
        ...
        generate an assignment "v := pv" and append it to Prolog[k];
        generate an assignment "v := v + c" and append it to Epilog[k];
        pv := v; k := k + 1
      END;
      r := pv
    END SubstituteRange;
```

(and the type of the procedure parameter r must be adjusted). When generating the n slicing loops, of course the vector length register must not be set and the innermost loop counter must be decremented in steps of 1.

As an example, the transformations applied to a 2-dimensional ALL statement are considered. After compilation of the range declaration and processing of the enclosed assignment, the syntax tree shown in Figure 3.7 is obtained.

```
VAR
  A, B: ARRAY 100, 100 OF REAL;
  a, b: INTEGER;
  x: REAL;
...
ALL i = 10 .. 20, j = a .. b DO
  A[i, j] := B[j, i] * x
END
```

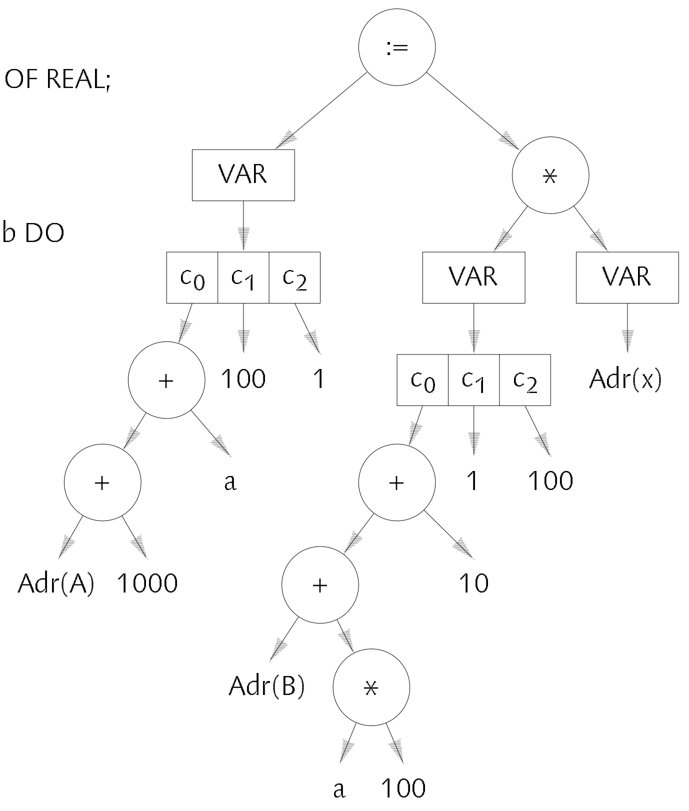


Figure 3.7 Syntax Tree of an Assignment within a 2-dimensional ALL Statement

The syntax tree is traversed recursively with SubstituteRange and the 2-dimensional range objects are substituted by 1-dimensional range objects. The resulting syntax tree and the prolog and epilog assignment statements are shown in Figure 3.8.

Prolog Assignments

k = 1: v_1 := Adr(A) + 1000 + a; w_1 := Adr(B) + a*100 + 10

k = 2: v_2 := v_1 ; w_2 := w_1

Epilog Assignments

k = 1: v_1 := v_1 + 100; w_1 := w_1 + 1

k = 2: v_2 := v_2 + VL; w_2 := w_2 + VL*100

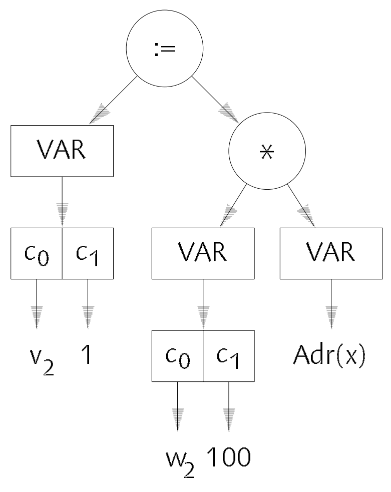


Figure 3.8 Syntax Tree after Transformation

The newly introduced variables required for the range object on the left-hand side are called v_k, the variables introduced for the range object on the right-hand side are called w_k. After introducing 2 additional counter variables x_1 and x_2, the prolog and epilog assignments as well as the transformed syntax tree can be used directly to generate two nested loops:

```
v1 := Adr(A) + 1000 + a; w1 := Adr(B) + a*100 + 10; (* prolog 1 *)
x1 := 11;
WHILE x1 > 0 DO
  v2 := v1; w2 := w1; (* prolog 2 *)
  x2 := b – a + 1;
  WHILE x2 > 0 DO
    VL := (x2 – 1) MOD VLmax + 1;
    ALL r = 0 .. VL–1 DO
      A'[v2 + r] := B'[w2 + r] * x
    END;
    v2 := v2 + VL; w2 := w2 + VL*100; (* epilog 2 *)
    x2 := x2 – VL
  END;
  v1 := v1 + 100; w1 := w1 + 1; (* epilog 1 *)
  x1 := x1 – 1
END
```

A' and B' stand for the arrays A and B regarded as "flattened" 1-dimensional arrays; i.e. the addresses of A' and B' correspond to the addresses of A and B, and A' and B' are of type ARRAY 100*100 OF REAL. Since the length of the range r within the enclosed ALL statement is not greater than VL_{max}, the ALL statement can be directly translated into vector instructions and the following instruction sequence is obtained for the example:

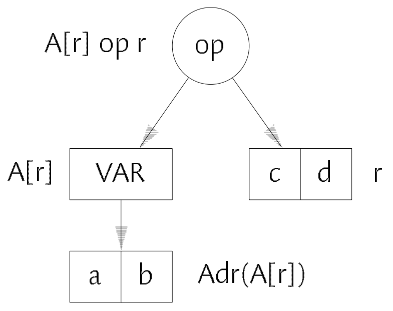
```
;         S0                  address of A
;         S1                  address of B
;         S2                  variable a
;         S3                  variable b
;         S4                  variable x
;         S5                  counter x1       (introduced by the compiler)
;         S6                  counter x2       (introduced by the compiler)
;         S7                  variable v1      (introduced by the compiler)
;         S8                  variable v2      (introduced by the compiler)
;         S9                  variable w1      (introduced by the compiler)
;         S10                 variable w2      (introduced by the compiler)
;
          S20 := 1000 + S0    ; Adr(A) + 1000
          S7 := S20 + S2      ; v1 := Adr(A) + 1000 + a
          S21 := 100 * S2     ; a*100
```

```
            S22 := S1 + S21              ; Adr(B) + a*100
            S9 := 10 + S22               ; w1 := Adr(B) + a*100 + 10
            S5 := 11                     ; x1 := l1
Loop1       S23 := S5 > 0                ; x1 > 0
            jump Exit1 (~S23)            ; WHILE x1 > 0 DO
            S8 := S7                     ; v2 := v1
            S10 := S9                    ; w2 := w1
            S24 := S3 – S2               ; b – a
            S6 := 1 + S24                ; x2 := l2 = b – a + 1
Loop2       S25 := S6 > 0                ; x2 > 0
            jump Exit2 (~S25)            ; WHILE x2 > 0 DO
            VL := S6                     ; VL := x2
  |         V26 := M[100, S10]           ; slice of B[j, i]
  |         V27 := S4 *F V26             ; slice of B[j, i]] * x
  |         M[1, S8] := V27              ; slice of A[i, j] := B[j, i]] * x
            S28 := VL                    ; get vector length of this iteration
            S8 := S8 + S28               ; v2 := v2 + VL
            S29 := 100 * S28             ; VL*100
            S10 := S10 + S29             ; w2 := w2 + VL*100
            S6 := S6 – S28               ; x2 := x2 – VL
            jump Loop2                   ; END
Exit2       S7 := 100 + S7               ; v1 := v1 + 100
            S9 := 1 + S9                 ; w1 := w1 + 1
            S5 := –1 + S5                ; x1 := x1 – 1
            jump Loop1                   ; END
Exit1       ...
```

3.7.3 Interesting Cases

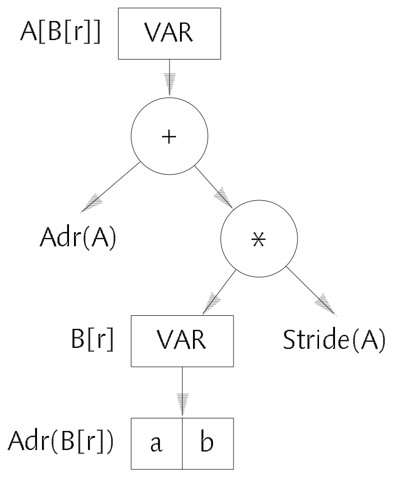
In this section, expression trees and corresponding code sequences for three more interesting cases are illustrated without going too much into the details. No additional complexity is introduced by these cases but only the previously described methods are consequently applied. For all examples a 1-dimensional range declaration introducing a range identifier r is assumed. A, B, C and D denote 1-dimensional real arrays of appropriate length. Coefficients are designated by small letters a, b, ... h. As explained in Section 3.7.2, the multi-dimensional case can always be reduced to the 1-dimensional case.

a) *Expressions of the form A[r] op r:*



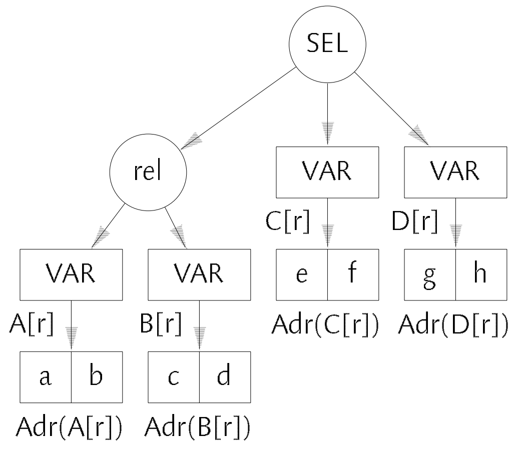
```
S10 := code for a
S11 := code for b
V12 := M[S11, S10]        ; A[r]
S13 := address of integer
vector 0, 1, 2 ... VLmax−1
V14 := M[1, S13]          ; 0, 1, 2 ... VL−1
S15 := code for d
V16 := S15 * V14          ; 0, d, 2d, ...
S17 := code for c
V18 := S17 + V16          ; r
V19 := V12 op V18         ; A[r] op r
```

b) *Expressions of the form A[B[r]] (without index checks):*



```
S10 := code for Adr(A)
S11 := code for a
S12 := code for b
V13 := M[S12, S11]      ; B[r]
S14 := code for Stride(A)
V15 := S14 * V13        ; B[r] * Stride(A)
V16 := M[V15, S10]      ; A[B[r]]
```

c) *Expressions of the form SELECT(A(r) rel B(r), C(r), D(r)):*



```
S10 := code for a
S11 := code for b
V12 := M[S11, S10]        ; A[r]
S13 := code for c
S14 := code for d
V15 := M[S14, S13]        ; B[r]
S16 := V12 rel V15        ; A[r] rel B[r]
S17 := code for e
S18 := code for f
V19 := M[S18, S17]        ; C[r]
S20 := code for g
S21 := code for h
V22 := M[S21, S20]        ; D[r]
V23 := V19 | V22 (S16)    ; SELECT(...)
```

3.8 Compilation of Array Constructors

Array constructors are used to construct subarrays which subsequently may be used as arguments corresponding to open array parameters. There are two kinds of array constructors, namely constructors that specify an *array variable* and constructors that specify an *array value*. The former may be used as arguments for variable parameters whereas the later apply as arguments for the predefined functions SUM and PROD only.

3.8.1 Array Variable Constructors

Firstly, array constructors that specify *array variables* are considered which may be used as arguments corresponding to *variable parameters*. Within a procedure, variable parameters stand for the variables passed as argument; i.e. they actually are *alias identifiers* for these variables. Usually variable parameters are implemented by passing the addresses of the variables only. If a subarray is to be passed, a *descriptor* specifying the addresses of the elements of the specified subarray must be passed. For *arbitrary* subarrays, such a descriptor would be an array containing the addresses of all the elements of the argument array:

```
VAR A: ARRAY 100 OF REAL;
...
PROCEDURE f (i: INTEGER): INTEGER;
...
PROCEDURE P (VAR a: ARRAY OF REAL; i: INTEGER);
BEGIN ... a[i] ...
END P;
...
P([i = 10 .. 19: A[f(i)]], 7)
```

Since f may denote an arbitrary function in the example, the descriptor to be passed would have to contain an array consisting of the 10 addresses (Adr(A[f(10)]), Adr(A[f(11)]), ... Adr(A[f(19)])). Within P, the element i could then be accessed by dereferencing the address i of the passed descriptor. While such an implementation would be feasible, it would probably not be desirable, due to the overhead implied by the introduction of such an address array on both sides, the caller of P and the callee P. Thus, a simpler solution is required.

In Oberon-V, the designators which may be used within an array variable constructor must contain linear range expressions only and no pointer dereferenciation must occur (cf. Section 3.5 and Appendix A). With this restrictions, passing an n-dimensional subarray requires passing an array descriptor containing *2n + 1* integers only as will be shown in the following.

An array variable constructor consists of an n-dimensional range declaration followed by a designator D denoting a variable. Without loss of generality one may write (cf. Section 3.4):

$$[\mathbf{r} = 0 \,..\, \mathbf{l}-1\text{: } D(\mathbf{r})]$$

The compilation proceeds as for an ALL statement without code generation (cf. Section 3.7). In a first step, the intermediate data structure is built, i.e. the range declaration is compiled and an expression tree is constructed for the designator D(**r**). Since all designators can be uniformly represented by *VAR nodes* (cf. Figure 3.2) and because only linear operations are involved when subscripting an array, the expression tree generated after parsing a designator containing only linear range expressions is simply a VAR node whose address is a *range object* containing n+1 coefficients (cf. Section 3.4 and 3.5). This range object represents a function $f \in AF^n(\mathbf{l})$ where **l** is the length of the range declaration.

Figure 3.9 illustrates this situation for an expression tree representing an array subscript i which is a linear range expression and thus can be represented by a range object. The construction of the expression tree proceeds as described in Section 3.3, but since linear operations over ranges are transformed into linear expressions over the ranges' coefficients, the resulting tree consists of a VAR node which address is again a range object. Note that also the address node of an array A itself (Adr(A)) could be a range object.

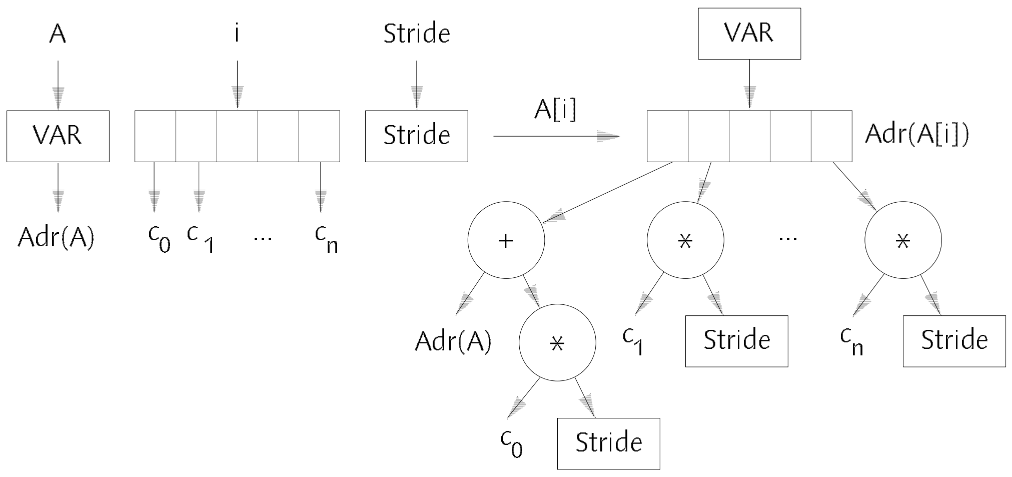


Figure 3.9 Expression Tree for a Linear Designator

According to its definition, the elements specified by the array constructor are the elements D(**i**) with **i** ∈ {**x** : 0 ≤ **x** < **l**}. The designator D(**r**) is represented by a VAR node whose address is a range object. This range object specifies an affine function f ∈ AF^n(**l**). For a particular **i** ∈ {**x** : 0 ≤ **x** < **l**}, f(**i**) is the address of the array element D(**i**). Thus, the addresses of all array elements D(**i**) are the values of f(**i**) for all **i**, i.e. the set ran(f) = {f(**i**) : 0 ≤ **i** < **l**}. Therefore, the subarray is clearly specified by this function f ∈ AF^n(**l**) which itself is specified by 2n+1 integers (cf. Section 3.2), namely its n+1 coefficients – which are the coefficients of the range object – and its length **l**. Apparently these 2n+1 integers are the parameters to be passed to the called procedure.

In the following example a 2-dimensional subarray – a diagonal plane – of a 3-dimensional array A is passed to a procedure P.

```
VAR A: ARRAY 10, 20, 30 OF REAL;
...
PROCEDURE P (VAR a: ARRAY OF ARRAY OF REAL);
   VAR i, j: INTEGER;
BEGIN ... a[i, j] ...
...
P([u, v = 0 .. 9: A[u, 2*v, 2*v + 1])
```

The range declaration introduces 2 range identifiers associated with 2 affine functions f_u, f_v ∈ AF^2(**l**) with **l** = (10, 10):

Function	*Denoted Range*
$f_u(\mathbf{x}) = (1, 0)\cdot\mathbf{x} + 0$	$\mathrm{ran}(f_u) = \{0, 1, \dots 9\}$
$f_v(\mathbf{x}) = (0, 1)\cdot\mathbf{x} + 0$	$\mathrm{ran}(f_v) = \{0, 1, \dots 9\}$

The designator A[u, 2∗v, 2∗v + 10] is translated into an expression tree in 3 steps according to Section 3.4 and 3.5. In each step, the expression tree consists of a VAR node at the root and a range object specifying a function $f_k \in AF^2(\mathbf{l})$ (it is assumed that Adr(A) = a_0 and Size(A[0, 0, 0]) = 1):

Designator	*Associated Function*	
A	$f_0 = a_0$	= Adr(A)
A[u]	$f_1 = (600, 0)\cdot\mathbf{x} + a_0$	$= f_0 + f_u * 600$
A[u, 2∗v]	$f_2 = (600, 60)\cdot\mathbf{x} + a_0$	$= f_1 + 2*f_v * 30$
A[u, 2∗v, 2∗v + 1]	$f_3 = (600, 62)\cdot\mathbf{x} + (a_0 + 1)$	$= f_2 + (2*f_v + 1) * 1$

The subarray argument is clearly specified by f_3 and **l**. Therefore, the values to be passed to P are the coefficients of f_3, (a_0 + 1, 600, 62) and the length **l** = (10, 10). Within the procedure P, the address of an element a[i, j] is computed by simply applying f_3 to i and j: Adr(a[i, j]) = f_3(i, j). The length **l** is necessary for index checks within P.

Unfortunately this translation scheme is not yet complete; there are two points which deserve special attention: Firstly, it must *not* be possible to pass subarrays that are larger than the subscripted array, i.e. some kind of *index check* is required; and secondly, as will be shown below, without further restrictions it is now possible to construct subarrays which violate a major property of arrays, namely the invariant that an array of length n consists of n *distinct* elements. For index checks the reader is referred to Section 3.9. The second problem is illustrated by a small example. Let A be an array of which a subarray is to be passed to a procedure P:

```
VAR
  A: ARRAY 5 OF REAL;

PROCEDURE P (VAR a: ARRAY OF ARRAY OF REAL);
...
P([i = 0 .. 2, j = 0 .. 1: A[i + 2∗j]])
```

The array constructor contains a 2-dimensional range declaration with length **l** = (3, 2); i.e. an array containing 3∗2 = 6 elements is constructed of a 1-dimensional array of length 5. Since no index range is violated, two elements of the newly constructed array must be *identical*! Figure 3.10 illustrates the pathological array mapping implied by the array constructor.

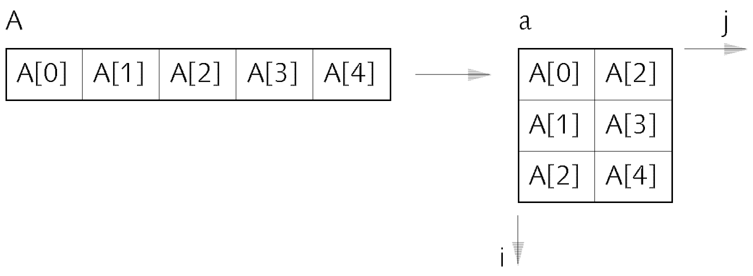


Figure 3.10 Pathological Array Mapping

Within the procedure P, both elements a[0, 1] and a[2, 0] denote *the same* element A[2]! In fact, an assignment to the variable a[0, 1] would change a[2, 0] too. It is easy to construct even worse situations; e.g. arrays where all elements are mapped to a single variable (using a constructor of the form [r = a .. b: v], where the range identifier r is not used within the designator v). Thus, a method is required to detect – or better – to avoid such pathological arrays. As explained before, a general array constructor consisting of an n-dimensional range declaration and a linear designator

[**r** = 0 .. **l**–1: D(**r**)]

defines an affine function $f \in AF^n(\mathbf{l})$ which specifies the mapping of the array indices to the corresponding element addresses. The mapping leads to a *pathological array* if the function f is *not injective*, i.e. if there are two different indices **x** and **y** with $0 \le \mathbf{x}, \mathbf{y} < \mathbf{l}$ such that both are mapped to the same address. After some arithmetics one gets:

$$\begin{array}{llll} & f(\mathbf{x}) & = f(\mathbf{y}) & (0 \le \mathbf{x}, \mathbf{y} < \mathbf{l}) \wedge (\mathbf{x} \ne \mathbf{y}) \\ & \mathbf{c}\cdot\mathbf{x} + c_0 & = \mathbf{c}\cdot\mathbf{y} + c_0 & \\ & \mathbf{c}\cdot\mathbf{x} - \mathbf{c}\cdot\mathbf{y} & = 0 & \\ & \mathbf{c}\cdot(\mathbf{x} - \mathbf{y}) & = 0 & \\ \text{with } \mathbf{z} = \mathbf{x} - \mathbf{y}: & \mathbf{c}\cdot\mathbf{z} & = 0 & (-\mathbf{l} < \mathbf{z} < \mathbf{l}) \wedge (\mathbf{z} \ne 0) \end{array}$$

The last equation $\mathbf{c}\cdot\mathbf{z} = 0$ is a *linear diophantine equation*, i.e. a linear equation in n *integer* variables. The trivial solution $\mathbf{z} = \mathbf{0}$ is not allowed, because $\mathbf{x}$ must be different from $\mathbf{y}$. Although methods exist to solve such equations (cf. [Banerjee 1988]), they are not considered further, because they had to be applied at run time in general, since the coefficients of f may be expressions containing variables. In Oberon-V, the problem is circumvented by three additional restrictions which apply within array variable constructors:

1. Within a *single* index expression of the designator in an array variable constructor, no two *different* range identifiers must occur.
2. If the array constructor contains an n-dimensional range declaration, all n range identifiers must be used at least once in the designator.
3. None of the range expressions must denote a constant value (e.g. $r*0 + c$, where r denotes a range, and c stands for a constant scalar value).

A linear range expression which contains no two different range identifiers is a range expression which corresponds to a function $f \in AF^n(\mathbf{I})$ where only a single coefficient c_k with $1 \le k \le n$ is not zero; i.e. f is a function of the form:

$$f(x_1, x_2, \dots x_n) = c_0 + 0x_1 + 0x_2 + \dots + c_k x_k + \dots 0x_n = c_0 + c_k x_k$$

and k corresponds to the position of the range identifier in the range declaration from left to right (due to the way range declarations are compiled, cf. Section 3.4). Together with restriction (3), this implies that c_k must not be zero, otherwise the range expression would be constant = c_0. Thus, the function $f(x_k) = c_0 + c_k x_k$ with $c_k \neq 0$ is *injective*.

In a designator of an array variable constructor, expressions can occur only as subscript expressions of arrays, due to the syntax of designators. Each expression denotes either a scalar value or it is a linear range expression corresponding to an injective function of the form $f(x_k) = c_0 + c_k x_k$. The range ran(f) denoted by such a function f specifies the indices of the elements of the subscripted array, and because each index occurs only once for all values of x_k ($0 \le x_k < l_k$), each array element is denoted only once in this array dimension. This must hold for all expressions which contain an arbitrary range identifier r_k ($1 \le k \le n$) of the range declaration. Since *all* range identifiers must be used because of restriction (2), it is not possible to denote the same array element more than once (if not all range identifiers would have to be used, an array variable constructor of the form [u, v = 0 .. 9: A[u]] would be possible, and for different v always the same element A[u] would be denoted). The situation is illustrated by means of the previous example:

```
VAR A: ARRAY 10, 20, 30 OF REAL;
...
[u, v = 0 .. 9: A[u, 2*v, 2*v + 1]
```

Each expression of the designator in the array constructor contains only a single range identifier u or v and the functions corresponding to the range expressions are:

Expression	*Function*	*Range*
u	$f_1(\mathbf{x}) = (1, 0)\cdot\mathbf{x} + 0$	$\mathrm{ran}(f_1) = \{0, 1, \ldots 9\}$
2*v	$f_2(\mathbf{x}) = (0, 2)\cdot\mathbf{x} + 0$	$\mathrm{ran}(f_2) = \{0, 2, \ldots 18\}$
2*v + 1	$f_3(\mathbf{x}) = (0, 2)\cdot\mathbf{x} + 1$	$\mathrm{ran}(f_3) = \{1, 3, \ldots 19\}$

For all three functions only a single c_k (k = 1, 2) is not zero, namely c_1 in f_1 (because u is the first range identifier in the range declaration) and c_2 in f_2 and f_3 (because v is the second range identifier in the range declaration). Let us consider f_3: now it is impossible that for different values of x_2 the same value is assumed by $f_3(x_1, x_2)$. Or from the view of the range expressions: it is impossible that the same value is assumed by the range expression 2*v + 1 for different values assumed by v (note that v assumes all values within its associated range). Since similar conditions are fulfilled by all range expressions and since all range identifiers are used, different array elements are denoted for different values of the range identifiers u and v.

The additional restrictions impose no problems in practical applications. Even checking whether the conditions are fulfilled or not is quite easy, although it might not seem so at a first sight. Restriction (1) is checked whenever an array index expression is a linear range expression: if so, the linear range expression is represented by a range object, and only a single coefficient of this range object must not be zero. Restrictions (2) and (3) are checked after compilation of the array constructor: if it denotes an array variable, the subarray is represented by a VAR node whose address is again a range object. If any of the coefficients c_k $(1 \leq k \leq n)$ of this range object is zero, either a range identifier has not been used or the range expression where it has been used is constant. Note that this check must be done at run time, if the coefficient expression is not a constant.

Altogether, the compilation of an array variable constructor proceeds as follows: the range declaration and the expression are compiled like ALL statements. However, if an index expression occurs which is a linear range expression (i.e. which is represented by a range object), restriction (1) is tested.

After processing the entire expression within the array constructor, the constructor denotes a legal subarray variable if the expression is represented by a VAR node whose address is a range object; otherwise an error message is raised. If any of the coefficients c_k ($1 \le k \le n$) of this range object is zero, an error message is raised. In the code generation phase, check code is generated for all coefficients which are not constant – they must not be zero (except c_0). Then, the n+1 coefficient expressions and the n lengths are evaluated and passed as arguments to the procedure. If the type of the designator within the array constructor is an array too, possibly additional coefficients and lengths must be passed.

It might be noteworthy that in Oberon [Wirth 1988a] passing of open arrays is also permitted, however not in this general form. The main difference is that in Oberon no *explicit stride* between two consecutive array elements may be specified, as it is actually the case here: the stride in dimension k corresponds to the coefficient c_k of the function f. In Oberon it suffices to pass the base address of the argument array and its lengths. The strides are then computed from the lengths (cf. [Wirth 1992]).

3.8.2 Array Value Constructors

The use of array value constructors is restricted to arguments for the predefined *reduction functions* SUM and PROD. Since these functions are known to the compiler, inline code can be generated for them and no sophisticated parameter passing mechanism is necessary. Thus, the problem of code generation for array value constructors is actually the problem of code generation for SUM and PROD. In the following the function SUM is considered only; similar considerations apply to PROD too. It is assumed that the summation order can be changed appropriately because addition is a commutative operation. Unfortunately, for floating-point addition this is not true and the result may be inaccurate especially if the operands differ largely in magnitude. If the summation order cannot be changed for reasons of accuracy, vectorization is not possible without specialized reduction instructions. In this case, an ordinary loop must be used instead of SUM. A general SUM call may be formulated as follows:

SUM([**r** = 0 .. **l**–1: E(**r**)])

where **r** = 0 .. **l**–1 stands for an n-dimensional range declaration and E(**r**) for an arbitrary expression possibly containing the range identifiers **r** = r_1, r_2, ... r_n. Roughly speaking, the same transformations can be applied to the expression tree corresponding to E(**r**) as to syntax trees of ALL statements (cf. Section 3.7). Let us consider the following *incomplete* ALL statement with the same range declaration **r** = 0 .. **l**–1 and the same expression E(**r**) like in the SUM call above:

```
ALL r = 0 .. l–1 DO
   ... := ... + E(r)
END
```

For an ALL statement, these transformations yield the necessary information required to construct a nest of n loops. For the incomplete ALL statement the following loop nest is obtained (noted in an informal manner):

```
loop_1
   ...
      loop_n
         set VL;
         ALL r' = 0 .. VL–1 DO
            ... := ... + E'(r')
         END
      end
   ...
end
```

The ALL statement can be completed by introducing an auxiliary array S of length VL_{max}:

```
ALL r = 0 .. VL_max–1 DO
   S[r] := 0
END;
loop_1
   ...
      loop_n
         set VL;
         ALL r' = 0 .. VL–1 DO
            S[r'] := S[r'] + E'(r')
         END
      end
   ...
end;
s := 0; i := VL_max;
WHILE i > 0 DO i := i–1; s := s + S[i] END
```

Obviously, the execution of the n nested loops computes VL partial sums in the array S (note that always VL ≤ VL_{max}). Thus, the desired total sum s is obtained by adding these partial sums at the end. The auxiliary array S can always be held in a vector register, thus speeding up the computation significantly. Note that this translation scheme relies on vector instructions that leave unused vector elements untouched (for VL < VL_{max}). If the vector computer at hand does not fulfill this requirement, a more complicated code pattern must be generated. As an example an instruction sequence implementing the dot product **x·y** = SUM([i = 0 .. n–1: x[i] ∗ y[i]]) for two real arrays x and y is shown:

```
;        S0                   address of x
;        S1                   address of y
;        S2                   length n
;        S3                   counter i              (introduced by the compiler)
;        S4                   slice pointer x↑       (introduced by the compiler)
;        S5                   slice pointer y↑       (introduced by the compiler)
;        V6                   auxiliary vector S     (introduced by the compiler)
;        S7                   auxiliary variable s   (introduced by the compiler)
;
         V6 := 0              ; initialize S
         S4 := S0             ; initialize x↑
         S5 := S1             ; initialize y↑
         S3 := S2             ; initialize i
Loop1    S10 := S3 > 0        ; i > 0
         jump Exit1 (~S10)    ; WHILE i > 0 DO
         VL := S3             ; set vector length
|        V11 := M[1, S4]      ; slice of x[i]
|        V12 := M[1, S5]      ; slice of y[i]
|        V13 := V11 ∗F V12    ; slice of x[i] ∗ y[i]
|        V6 := V6 +F V13      ; partial sum
         S14 := VL            ; get vector length of this iteration
         S4 := S4 + S14;      ; x↑ := x↑ + VL
         S5 := S5 + S14;      ; y↑ := y↑ + VL
         S3 := S3 – S14       ; i := i – VL
         jump Loop1           ; END
Exit1    S7 := 0              ; s := 0
         S3 := VLmax          ; i := VLmax
Loop2    S15 := S3 > 0        ; i > 0
         jump Exit2 (~S15)    ; WHILE i > 0 DO
         S3 := –1 + S3        ; i := i–1
         S16 := V6[S3]        ; S[i]
         S7 := S7 +F S16      ; s := s + S[i]
         jump Loop2           ; END
Exit2    ...                  ; result of SUM(...) in S7 (s)
```

On a real machine, e.g. a Cray Y-MP [Cray 1988], this computation is completely dominated by the vector instructions (marked with a vertical bar on the left margin). Due to its two memory ports and its independent functional units, two vector elements can be loaded, multiplied and added in a single clock cycle by the Cray Y-MP. The scalar operations updating the slice pointers and the loop counter are executed "in the wake of the vector instructions"; i.e. by independent scalar functional units operating while the vector instructions are executed. For long vectors ($n \geq 1000$), the execution time for a dot product is approximately n clock cycles. With a cycle time of 6ns, it is possible to obtain an execution time of approximately 6ms for the dot product of two vectors of length $n = 10^6$.

3.9 Index Checks

Index checks are required to guarantee that only array elements within the index bounds of an array are accessed; i.e. an index expression must evaluate to a positive integer smaller than the length of the subscripted array.

For index expressions denoting a scalar value, the index check can be performed at compile time if both the index is a constant value and the length of the array is known at compile time, i.e. if the array is not an open array. In all other cases, an index check must be performed at run time. With the unified representation of designators (Section 3.3), array accesses are not directly recognizable anymore within the expression tree. A simple solution are explicit *index check nodes*: an index check node has two operands, namely the index expression and the length of the array (Figure 3.11.a). The result of an index check operation is the checked index. In the code generation phase of a compiler, an index check node is translated into a code sequence which raises a trap if the index violates its index bounds (zero and the array length minus one). Figure 3.11.b shows the expression tree with index check node corresponding to the designator A[i].

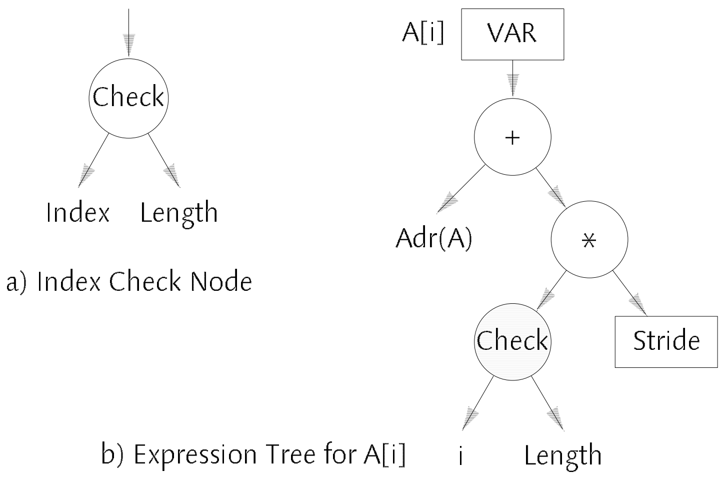


Figure 3.11 Index Check Nodes

For index expressions that are linear range expressions, i.e. that are represented by range objects, the problem is more complicated. Since a range object denotes not only a single value but a set of values, a correct index check must test whether *all* these values are within the allowed index bounds. Note that a simple insertion of an index check node as in the scalar case would disable the transformation of linear expressions over ranges to linear expressions over the ranges' coefficients (Section 3.5).

As explained in Section 3.4, a range object corresponds to an affine function $f \in AF^n(\mathbf{l})$ within the scope of an n-dimensional range declaration of length $\mathbf{l}$. The range object stands for all values ran(f). Thus, if the range object is used as index of an array with length L, all values of ran(f) must be positive and smaller than L; i.e. the condition $(0 \leq \min(f)) \wedge (\max(f) < L)$ must hold. Because $f \in AF^n(\mathbf{l})$, using Theorem 3.2 (cf. Section 3.2) the minimum is $\min(f) = c_0 - \mathbf{c}^- \cdot (\mathbf{l}-1)$ and $\max(f) = c_0 + \mathbf{c}^+ \cdot (\mathbf{l}-1)$. In summary, for a given range object containing the coefficients $c_0, c_1, \ldots c_n$ and representing a function $f \in AF^n(\mathbf{l})$, an index check must assert that

$$(0 \leq \min(f)) \wedge (\max(f) < L)$$

with $\quad \min(f) = c_0 - \mathbf{c}^- \cdot (\mathbf{l}-1)$
and $\quad \max(f) = c_0 + \mathbf{c}^+ \cdot (\mathbf{l}-1)$

where $\mathbf{l}$ is the length of the range declaration and L is the length of the subscripted array. Let us consider the following example:

```
VAR
   A: ARRAY 100 OF REAL;
...
ALL u = –4 .. 9, v = 0 .. 5, w = 10 .. 20 DO
   ... A[3*w – 2*v + u] ...
```

Within the ALL statement the array A is subscripted by a linear range expression. The functions f_u, f_v and f_w corresponding to the ranges associated with the range identifiers u, v and w as well as the function f corresponding to the linear range expression are:

Function	
$f_u(\mathbf{x}) = (1, 0, 0)\cdot\mathbf{x} + (-4)$	
$f_v(\mathbf{x}) = (0, 1, 0)\cdot\mathbf{x} + 0$	
$f_w(\mathbf{x}) = (0, 0, 1)\cdot\mathbf{x} + 10$	
$f(\mathbf{x}) = (1, -2, 3)\cdot\mathbf{x} + 26$	$(= 3f_w - 2f_v + f_u)$

and the length vector is $\mathbf{l} = (14, 6, 11)$. To check whether the range expression violates the index bounds of the array A or not, it is necessary to compute min(f) and max(f). With $\mathbf{c} = (1, -2, 3)$, $c_0 = 26$ and Theorem 3.2 the following values are obtained:

$$\min(f) = c_0 - \mathbf{c}^{-}\cdot(\mathbf{l}-1) = 26 - (0, 2, 0)\cdot(13, 5, 10) = 16$$
$$\max(f) = c_0 + \mathbf{c}^{+}\cdot(\mathbf{l}-1) = 26 + (1, 0, 3)\cdot(13, 5, 10) = 69$$

Because $(0 \le 16) \wedge (69 < 100)$ the index bounds are not violated by the range expression. In general, this computation cannot be performed *completely* at compile time, since either the coefficients c_k $(0 \le k \le n)$ or the lengths l_k $(1 \le k \le n)$ may not all be constant. However, note that *partial index checks* are often possible, since for the computation of min(f) only the lengths l_k are required for which the corresponding c_k is negative (for positive c_k's the value c_k^{-} is zero and also $c_k^{-}(l_k-1) = 0$, independent of the value of l_k). A similar relationship holds for max(f) and positive c_k's.

If an index check cannot be performed *completely* at compile time, it should be performed at run time. As illustrated in the example before, a range object is only *valid* as array index, if it fulfills the *constraint* $(0 \le \min(f)) \wedge (\max(f) < L)$ imposed by the index check for the function f specified by the range object. Such a constraint can be represented by a *constraint node* associated with a range object (Figure 3.12).

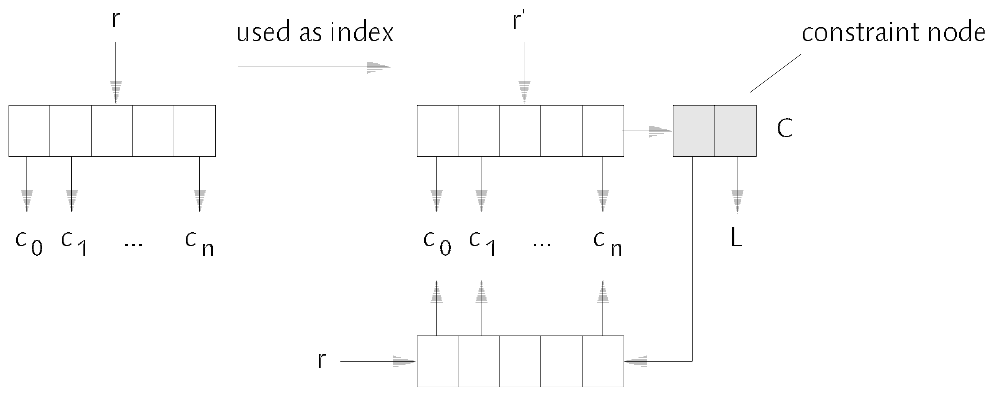


Figure 3.12 Range Constraints

Thus, if a range r is to be used as an index and the index check cannot be performed completely at compile time, the range r is copied and a constraint node C is associated with its copy r'. It is vital to copy r, because it might be shared by different expression trees (cf. Section 3.5). The constraint node contains a reference to the *constrained range* r and thus to the coefficients of the specified function f as well as a reference to the length L of the subscripted array. The length **l** of the range declaration is known within the scope of the declaration and is always the same for all range objects. Hence, it is not necessary to reference it by the constraint node.

If a range r is used within a range expression E(r), possibly producing a new range object r', a constraint node C of r must be adopted, because the range object r' representing E(r) is also valid only, if the constraint(s) of its operand r are observed. If two possibly constrained range objects r_1 and r_2 are added, the resulting range object r' must obey both the constraint(s) C_1 of r_1 and C_2 of r_2. Thus, a range object may be associated with an entire list of constraints to be observed (Figure 3.13). Such constrained range objects occur only, when subscripting arrays. As explained earlier (Section 3.5), subscripting an n-dimensional array A with n ranges r_1, r_2, ... r_n, A[r_1, r_2, ... r_n], implicitly leads to an expression $e = a_0 + r_1 * s_1 + r_2 * s_2 + r_n * s_n$ where a_0 is the address of A and the s_k's denote the strides in dimension k. Assumed that index checks cannot be performed at compile time, for each index r_k a new constraint node C_k is generated. Adding all terms of the expression e leads again to a range object, and this range object is associated with the list of all these constraint nodes C_k ($1 \leq k \leq n$). Note that also constraint nodes must be copied sometimes, because they may be shared by different range objects: in Figure 3.13 the constraint node C_1 must be copied (C_1') for the constraint list of r', whereas the constraint node C_2 may be shared.

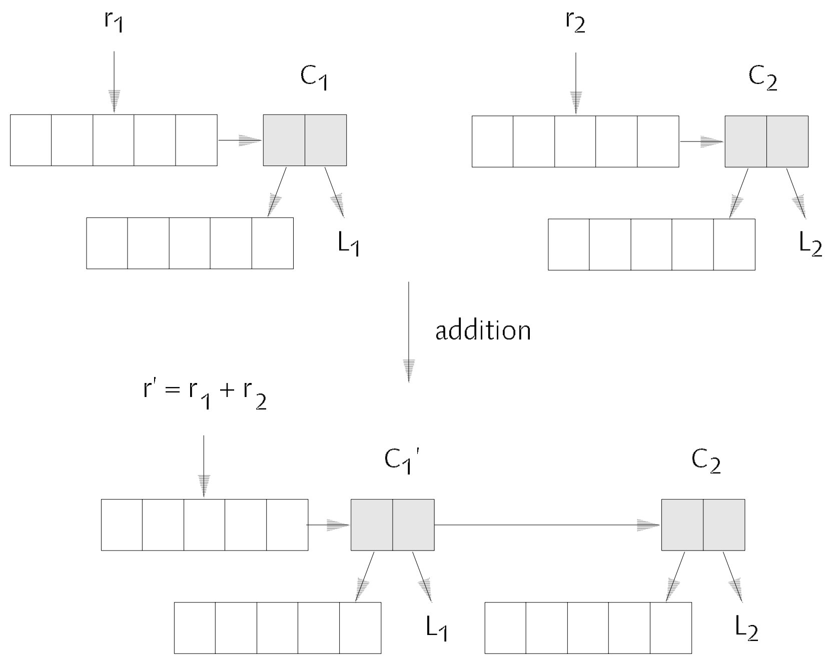


Figure 3.13 Addition of Constrained Ranges

Thus, after processing an ALL statement or an array constructor, each range object within the expression tree(s) may have a list of constraint nodes associated with it which denote the range constraints to be observed. In case of an ALL statement, all these constraint nodes are simply collected during the traversal of the ALL statement (e.g. as a first step in the SubstituteRange procedure, cf. Section 3.7.2) and held in an additional list of constraint nodes associated with the ALL statement. In the code generation phase, first index check code is generated for each of these constraint nodes (using Theorem 3.2). Then code is generated for the ALL statement, as explained in Section 3.7. Note that this approach is not correct if array elements are accessed under certain conditions only, e.g. in conditional expressions. In this case no constraint node must be associated with the range object in question but an index check node is introduced instead. Since no vector instructions but scalar code only can be generated for conditional expressions containing *and* and *or* operations, the index check node also can be translated conventionally. The arguments of the predefined function SELECT are always evaluated unconditionally.

In case of a legal array variable constructor, only a single range object remains after processing the array constructor. If this range object is constrained, in the code generation phase first index check code is generated for the constraints, and then parameter passing code is emitted as explained in Section 3.8.

4 Related Work

This chapter comprises a survey of other programming languages which are especially concerned with array handling or vector processing. This survey must necessarily be incomplete and merely should give some idea of other approaches but is not intended to cover these languages in detail. Fortran 77 and Fortran 90 are not mentioned here, they have been discussed in Chapter 1.

CFD. In order to make better use of the Illiac IV hardware, the programming language CFD was developed at NASA Ames Research Center [Stevens 1975]. CFD allows the programmer to fully control the underlying features of the hardware. No specific attempt was made to keep the language machine independent. The language itself strongly resembles Fortran but provides specific vector operations driving the 64 processing elements (PE) of the Illiac IV. Expressions are translated to PE instructions which operate on 64 elements even if only scalar values are involved. Thus, the programmer is forced to "think parallel" if maximal performance is desired. Two compilers were built, one translating CFD into relocatable machine code for the Illiac IV and one translating CFD into Fortran. CFD is one of the first languages designed explicitly for array processors. From the programmer's point of view, CFD provides operations on vectors that always have length 64. Operations on longer vectors have to be coded manually. With respect to this, CFD is comparable to SL/1 (see below).

IVTRAN. IVTRAN is an extension of Fortran designed for the Illiac IV computer [Erickson 1975]. Normally IVTRAN programs are produced by the Fortran compiling system for the Illiac IV [Millstein 1975] and hence serve as a kind of intermediate program representation during Fortran compilation. IVTRAN contains all features of "standard Fortran" (i.e. an amalgam of IBM and CDC Fortrans in 1975) and a few extensions in order to efficiently use the features of the underlying hardware. The DO FOR ALL statement is used to specify parallel operations on array elements. For example, using DO FOR ALL the following statements compute the square roots of the absolute values of all elements of a 3 x 7 x 10 array A:

```
      DO 10 FOR ALL (I, J, K) / [1 ... 3] .C. [1 ... 7] .C. [1 ... 10]
         IF (A(I, J, K) .LT. 0.0) A(I, J, K) = –A(I, J, K)
         A(I, J, K) = SQRT(A(I, J, K))
10    CONTINUE
```

Assignment statements enclosed within DO FOR ALL statements are executed *in parallel* for all values specified by the *index set*. The index set is the set of all n-tuples of integers specified in the header of the DO FOR ALL statement (in our example the index set comprises the 3-tupels (I, J, K) with I ∊ {1, 2, 3}, J ∊ {1, 2, ... 7} and K ∊ {1, 2, ... 10}). This stands in contrast to Oberon-V, where the assignments enclosed in ALL statements may be executed in parallel but are not required to do so.

The Illiac Fortran compiling system comprises three sections: the "paralyzer", the compiler and a support package. The paralyzer analyzes conventional Fortran programs and tries to reveal potential parallelism by inspecting the innermost DO loops. As a result an IVTRAN program is produced using the special IVTRAN features if possible. In a next step the IVTRAN program is compiled in a rather traditional fashion and Illiac object files are generated. IVTRAN is the ancestor of many of todays Fortran dialects which are enriched by new features in order to give the programmer the possibility to specify parallelism in programs.

Vienna Fortran. Vienna Fortran [Zima et. al. 1992] is an experimental extension of Fortran 77 [Brainerd 1978] oriented towards the programming of massively parallel machines with distributed memory. One of the main problems occuring when programming such machines is the distribution of arrays to different processing nodes. Vienna Fortran provides special mechanisms to describe such array layouts and to compute the addresses of the processing nodes which hold particular array elements or subarrays. Despite the conventional Fortran 77 statements and some Fortran 90 extensions [Metcalf 1987], Vienna Fortran provides a FORALL loop which is similar to the ALL statement in Oberon-V. A FORALL loop "allows the user to assert that different instantiations of the loop body are independent and can be logically executed in parallel"; the independence is not checked by the compiler. On a multi-processor architecture, each "iteration" of the loop can be executed on another processor. Since the execution of a loop iteration must not change any data items used by another iteration, all non-local data used by a particular iteration can be gathered before the start of the iteration. Similarly, non-local data items computed in a particular iteration can be communicated back to other processors after the end of the iteration. This allows the compiler to optimize the communication overhead required for the loop. Thus, the emphasis lies in the parallelization of statements by distribution. Currently the language is still under development and only research compilers are available.

APL. The programming language APL, sometimes also called "Iverson Notation" is based on the work of K. E. Iverson [Iverson 1962]. APL is a highly concise language designed to deal with arrays which are the only data structure available (some newer APL implementations also provide other data structures like *packages* [Berry 1979]). Scalar values are considered as arrays with rank (dimension) zero. APL programs consist of a sequence of expressions and some kind of computed goto's; assignment statements are considered as expressions. Expressions have arrays as operands and evaluate to arrays. By defining new functions, a program may be structured.

APL is famous for its powerful built-in *functions* and *operators* which are denoted by special characters. All functions are at most dyadic and always right-associative. There are *scalar functions* and *non-scalar functions*: the former are defined on scalar arguments and operate element-wise when applied to arrays, the latter are defined on arrays only. The right-hand side of an assignment is always evaluated first and then assigned to the variable on the left-hand side. Derived functions may be built by using (built-in) functions as arguments of operators. Function arguments are also arrays. In order to pass a subarray, an appropriate function is used to retrieve the desired subarray which is then passed as argument (which requires to copy the array in general).

APL is defined considerably concise and highly consistent both syntactically and semantically. Certain implementations use mathematical properties of APL expressions in order to speed up execution [Abrams 1970]. APL was the first high-level language for array processing and had a significant impact on other languages and systems. The new Fortran standard Fortran-90 [Metcalf 1987] adopts the idea of applying scalar operations element-wise on arrays. The Matlab system [MathWorks 1987] is an interactive system that pursues quite a similar philosophy (while using a more conventional syntax). For non-array oriented applications however it seems difficult to use APL: expressions are probably too powerful but on the other hand control structures are missing. Furthermore, if no restrictions are applied, APL cannot be compiled completely (because of a special function which interprets a string argument at run-time) and hence an interpreter is required.

Glypnir. Glypnir is a programming language developed at the University of Illinois at Urbana-Champaign for the Illiac IV computer [Lawrie 1975]. Except that Glypnir's syntax is Algol-based, it is similar to CFD (see Section 4.1). In order to obtain code which is as efficient as possible, it was decided not to hide the machine architecture to the programmer but to provide full control over the hardware. The machine architecture is mainly reflected in two categories of variables, namely "CU variables" and "PE variables". The former hold scalar values which are to be manipulated by the *C*ontrol *U*nit of the Illiac IV. The latter denote vectors or "swords" containing 64 elements to be processed by one of the 64 *P*rocessing *E*lements of the machine. Thus, expressions containing PE variables denote expression on vectors of length 64. Operations on longer vectors must be translated manually into a sequence of operations on swords. Glypnir is block structured and each block may have its own local storage. Subroutines resemble Algol procedures but they cannot be recursive and their arguments are called by value.

SL/1. The programming language SL/1 was developed at NASA Langley Research Center for the CDC Star-100 computer [Basili 1975]. The design of the language was determined by a few strong design goals: the language should be as small as possible, it should be easy to implement in order to be able to develop a compiler with minimal effort, it should force the user to express algorithms such that the hardware is used efficiently, and it should be extensible if necessary. As a consequence, the basic data types available in SL/1 can be mapped directly to the underlying hardware and most language operations can be implemented by a single machine instruction of the Star-100 computer. Beside of the scalar types (short) *real*, (short) *integer* and *logical* the corresponding vector types (short) *real vector*, (short) *integer vector* and *logical vecto*r were defined and considered as arrays consisting of a fixed number of elements of the specified scalar type. The only structured type which can be defined by the programmer is the (multi-dimensional) array, with the restriction that each element must be a vector or a string (a vector of dynamic length). The set of available operators is considerably large and consists of almost all operations that can be translated directly into a single machine instruction plus whatever is necessary to handle SL/1 data types. As far as possible single character symbols were chosen for operators. The fact that the Star-100 hardware offers an instruction to store elements of a vector depending on bits of a bit vector lead to a tryadic assignment statement:

```
(C, Z) := A + B
```

where C is the result vector and Z a bit vector controlling the assignment of individual vector elements. The available control structures in SL/1 are more conventional and include the WHILE, REPEAT, CASE, IF and FOR statement. The approach used in the design of SL/1 is quite radical and leads to a language that is inherently hardware dependent. In some sense SL/1 can be regarded as high-level assembler language.

Actus. The programming language Actus [Perrott 1979, 1983] is an offspring of the Pascal language family [Wirth 1971]. Actus is one of the first modern languages explicitly designed to ease array and vector processing. The key concept of the language is the notion of the *extent of parallelism* (eop): With each data structure based on an array and to most of the control structures an eop is associated. The eop denotes the set of array indices for which operations are to be applied in parallel. If a set of array elements is going to be processed in parallel, this must already be stated in the array declaration. For example, the variable declaration

```
VAR
  A: ARRAY [1 : 100, 1 .. 100] OF REAL;
```

introduces a matrix for which every column may be accessed in parallel, i.e. which has a (maximum) eop of 1:100. In the declaration the dimension in which the elements may be accessed in parallel is denoted with a colon; for some technical reason this is only possible for a single dimension. In general, A[1 : 100, j] will access the complete j-th column of this array, and A[i, 1 : 100] will access the complete i-th row. Such a column or row is then called a *parallel array*. All arithmetic operators are extended such that they accept scalar values but also parallel arrays; e.g. to scale all elements of the j-th column by a scalar value c the following expression is used

```
A[1 .. 100, j] * c
```

Obviously, the number of elements denoted by the two operands must be the same (with exception of scalar operands) otherwise an error occurs. Furthermore it is necessary that the eop's match. For example, the expression

```
A[1 .. 10, i] + A[2 .. 11, j]
```

is defined to be incorrect and must be rewritten to

```
A[1 .. 10, i] + A[1 .. 10 shift 1, j]
```

Two allignment operators *shift* and *rotate* are provided to handle this problem. Furthermore it is possible to declare index sets (to be used subsequently as subscripts for parallel arrays) and parallel constants similar to parallel arrays.

When a parallel array is used within a statement, the extent of parallelism of the array implicitly denotes the eop of the statement. Using the within statement, it is possible to specify the eop explicitly for an entire statement sequence. A hash mark (#) is used to denote the eop:

```
WITHIN 1 : 100 DO
  BEGIN
    A[#, i] := A[#, j] * c
  END
```

In case of an assignment statement, the expression on the right-hand side is evaluated and assigned *in parallel* for all array elements specified by the eop. The required dependence analysis is simplified by the conditions required for the eop's and the information provided by the shift and rotate operators. Some statements set the eop implicitly, for example the IF statement:

```
IF A[1 : 10, i] > 0 THEN A[#, i] := 0
ELSE A[#, i] := 1
```

The guard of the IF statement is evaluated and yields a set of indices for which the guard is true. This set is then used as the eop for the statements in the THEN part. The set of indices for which the guard yields false is used as the eop in the ELSE part. Similar rules are used for other control structures.

Parameter passing is extended such that it is possible to pass *conformant arrays*: the array bounds of formal parameters need not be known at compile time but are set at run time when passing the actual parameters. For example, a procedure P with heading

```
PROCEDURE P (VAR a: ARRAY [beg .. end, extent] OF REAL);
```

expects a two-dimensional (sub-)array argument. The identifiers *beg* and *end* take on the values of the first and last index in the first dimension of the passed array while *extent* takes on the eop of the second dimension (i.e. the rows of the passed array must be accessible as parallel arrays). The result type of a function may be a 1-dimensional array.

In contrast to Actus, in Oberon-V only independence of (assignment) statements can be expressed. This independence property allows for parallel execution but *does not require* it. In Actus, processing parallel data requires parallel execution or at least emulation of it.

Matlab. Matlab is "a high-performance interactive software package for scientific and engineering numeric computation" [MathWorks 1987]. Matlab (which stands for *matrix laboratory*) was originally written to provide easy access to matrix software packages such as Linpack [Dongarra 1979]. It evolved over several years and has become a standard tool in many areas. Matlab is an interactive system and its basic data structure is – as in APL – the array. Matrices may be entered and used directly in expressions without any form of declaration; hence Matlab acts like a desk calculator and is perfectly suited for experimenting. Powerful matrix decomposition algorithms are directly accessible by builtin functions. For example, the following statements compute the LU factorization of a 3x3 matrix A, assign its inverse A–1 to X and calculate its determinant in D (no output is shown here):

```
A = [1 2 3; 4 5 6; 7 8 9]
[L, U] = lu(A)
X = inv(A)
d = det(A)
```

Conventional control flow statements (such as a FOR statement) make Matlab a high-level matrix language. Using *script* and *function files* a program can be structured.

5 An Oberon-V Compiler

A compiler for a new language L serves both as a proof that an implementation of the language is feasible and as a tool for practical applications of L and hence the exploration of its usefulness. To be useful or *real* [Cardelli 1989], a language L should be theoretically and practically complete. While the former condition can be expressed as *Turing completeness*, i.e. the ability of the language to express every computable function, the latter condition is harder to quantify. Due to Cardelli, practical completeness is "the ability of a language to conveniently express its own (a) interpreter; (b) translator; (c) run-time support; (d) operating system". Since Oberon-V essentially provides the same features like Oberon [Wirth 1988a], and Oberon has been used to implement its supporting Oberon System [Wirth 1988b], it should be possible to fulfil even criterion (d) by Oberon-V. However, currently only a minimal form of run-time support has been implemented using Oberon-V.

The Oberon-V compiling system consists of the compiler *OV*, an object-file decoder and an interface browser. OV is a cross-compiler for the Cray Y-MP written in Oberon. It runs under the Oberon System and generates relocatable Cray object files. The decoder – a necessary prerequisite for a compiler writer embarking on the implementation of a back-end for a new target architecture – allows inspecting ordinary Cray object files. A dual role is played by the browser, which is used to extract interface descriptions à la Modula-2 [Wirth 1985] out of compiler generated symbol files (cf. Section 5.2.5). The browser was inspired by a similar tool in the Oberon System and proved to be quite helpful for programming [Templ 1989].

In this chapter the general outline of the OV compiler is described. Section 5.1 explains the module structure of the compiler. In Section 5.2 the intermediate data structure produced by the front-end is presented in detail. A rough survey of the internals of the back-end is given in Section 5.3. Finally, Section 5.4 summarizes the main trouble spots of the Cray Y-MP instruction set. Oberon program extracts have been translated to Obreon-V.

5.1 Modularization

The Oberon-V compiler is a two-phase compiler. In the first phase the source text is analyzed by a recursive-descent parser, and an intermediate data structure – a symbol table and associated syntax trees – representing the program is built. In the second phase the intermediate data structure is traversed and code is generated. Two disjoint sets of modules can be assigned to each phase: the *front-end* implements the first phase and is designed to be machine independent, whereas the low-level code generator is constituted by the *back-end* modules which are machine dependent by their very nature. Both parts are held together by the module OV which implements the user interface represented by a single command called *OV.Compile*. OV calls the front-end and passes the generated intermediate data structure, which is the only parameter, to the back-end. Since the front-end is machine independent, it may also be used either as a stand-alone application, e.g. for syntax and type checking of Oberon-V modules, or as a basis for other back-ends.

A portable Oberon-2 compiler (OP2) with a similar structure has been built before [Crelier 1990]. However, only relatively simple parts of the front-end could be reused for Oberon-V, namely the scanner and the parser. Due to differences in the language and a lower level of abstraction used in the intermediate data structure of the Oberon-V compiler, many parts had to be rewritten. Hence, it was decided to construct a new compiler from scratch; thereby applying the experience gained with OP2. Nevertheless, the overall structure of the two compilers OP2 and OV is much the same, with a notable exception: no machine-level virtual code is generated by OP2 (see below).

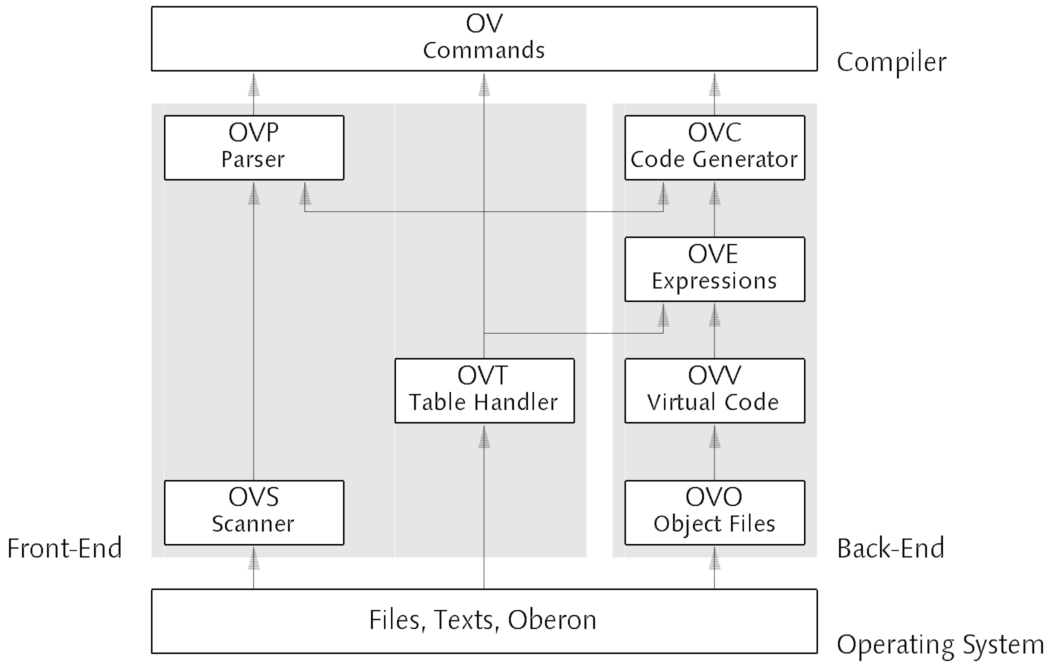


Figure 5.1 Modularization

Figure 5.1 illustrates the structure of the compiler, with the modules ordered according to their abstraction from bottom to top (with the highest abstraction on top) and separated into front-end and back-end.

A short description shall give a rough idea of the translation process of an Oberon-V program: A source text to be compiled is decomposed into a sequence of lexical units, called *symbols* or *tokens*, by means of the scanner *OVS*. The syntactic structure of the token stream is analyzed by the parser *OVP* which also builds the intermediate data structure. The required data types and procedures are provided by the table handler *OVT*. OVT acts also as an interface between front-end and back-end and hence is the only module that is imported by almost all other modules. When the source text has been analyzed without detecting any errors, the generated data structure is passed to the code generator *OVC*. OVC traverses the data structure and generates *virtual code* for a virtual machine by means of the procedures provided by *OVE*. This virtual code, handled by the module *OVV*, allows for some optimizations, namely *common subexpression elimination* and *instruction scheduling*. The virtual code is then translated into Cray machine code. Finally, the object code and some tables holding information for the linker are assembled and a relocatable Cray object file is generated (*OVO*).

5.2 Intermediate Program Representation

The main task of the front-end is to generate an intermediate program representation of a compilation unit. This intermediate data structure provides the basis for all the transformations explained in Chapter 3. In the following, it is not explained how this data structure is built from a textual representation of a program, but it is assumed that the reader is familiar with the standard techniques for scanning and parsing a program text according to its defining grammar (see e.g. [Wirth 1992], Chap. 12). It remains to say that the front-end module *OVP* is a simple recursive descent parser, which constructs the intermediate data structure by means of the procedures provided by module *OVT*.

The intermediate data structure consists of the symbol table information and associated syntax trees representing statement sequences of procedure and module bodies. The symbol table represents all declarations, i.e. all *identifiers* and their associated *objects* (cf. Section 5.2.1). Identifiers are only visible within their *scopes*. Since scopes are properly nested in Oberon-V, they are best managed using a *stack*. During parsing, the visible identifiers can be found by searching the scope stack, starting with the top scope. Using Oberon-V, the primary data structures and operations of the table handler OVT are described by the following definitions:

```
TYPE
  Scope = RECORD
    prev: ↑Scope;
    first: ↑Ident
  END;

  Ident = RECORD
    next: ↑Ident;
    name: ARRAY 32 OF CHAR;
    exported: BOOLEAN;
    bound: ↑Object;
    usage: INTEGER (* usage = 0 <=> not used *)
  END;

VAR
  universe: ↑Scope;

PROCEDURE NewScope (VAR scope: ↑Scope;  prev: ↑Scope);
PROCEDURE Insert (scope: ↑Scope; VAR name: ARRAY OF CHAR; VAR id: ↑Ident);
PROCEDURE This (scope: ↑Scope; VAR name: ARRAY OF CHAR): ↑Ident;
```

New scopes are generated by means of the procedure *NewScope*; the argument *prev* denoting the previous scope allows to build a stack of scopes. For every module, procedure, record or range declaration a new scope is opened. For simplicity all identifiers are held in a linear list of *Ident* nodes within a scope. With *Insert* a new identifier is appended to this list, and at the same time it is checked whether the identifier has been declared before in the same scope (and if so, an error message is raised). Each Ident node contains information about the *next* identifier in the list, the identifier *name*, whether this name be *exported* or not and which *object* is associated with or *bound* to the identifier. If an identifier is used to refer to a particular object, the identifier node and hence the associated object are retrieved using the *This* function. The *universe* scope is used as the predecessor of all module scopes; it contains a collection of predefined objects.

The *usage* field provides a simple heuristic measure for the *dynamic frequency of usage* of an identifier and may be used to allocate frequently accessed variables in registers (cf. Section 5.3.1). Whenever an identifier is used in the program text, its usage field is incremented taking the current nesting level of its surrounding loops into account (and also IF, CASE and ALL statements). The value zero indicates that the corresponding identifier has not been used at all within its defining procedure or module.

5.2.1 Representation of Objects

In Oberon-V there are constant, type, variable, procedure and range declarations. In each declaration an appropriate object is introduced and associated with an identifier. These *special* objects are naturally expressed as *type extensions* of the base type *Object*. In contrast to some Oberon compilers, where only a single object type is used to represent a similar variety of objects, i.e. where the information of all kinds of objects is "flattened" and packed into a single record type [Wirth 1992][Crelier 1990], using type extensions has the advantage of being more readable and secure. The additional assertions (type guards) required in the compiler source provide valuable information for the programmer who is interested in the correctness of the implementation, but they do not seem to slow down execution significantly. It is convenient to introduce a few additional objects, namely objects representing parameters, record fields, predefined (standard) procedures and modules (*Address* and *Range* objects as well as *Stat* nodes are introduced later). All objects have a type (*typ*) associated with them, since Oberon-V is a strongly typed language (cf. Section 5.2.2).

```
TYPE
   Object = RECORD
      typ: ↑Struct
   END;

   Const = RECORD (Object)
      int: INTEGER;
      re, im: REAL;
      ...
   END;

   Type = RECORD (Object) END;

   Var = RECORD (Object)
      adr: ↑Object
   END;

   Par = RECORD (Var);
      var: BOOLEAN;
      parAdr: ↑Address
   END;

   Field = RECORD (Object)
      offs: ↑Const
   END;

   Proc = RECORD (Object)
      body: ↑Stat;
      adr: ↑Address
   END;

   StdProc = RECORD (Object)
      f, nofPars: INTEGER
   END;

   Module = RECORD (Proc)
      fingerprint: INTEGER
   END;
```

New objects are created and initialized by a set of initialization procedures (not shown here). Parameters (*Par*) are regarded as *special variables* (*Var*). They may be either variable or value parameters, determined by the value of the field *var*. Besides the normal variable address (*adr*), parameters also have a parameter address (*parAdr*), which corresponds to the address seen by a caller of the corresponding procedure (see also Figure 5.4). The position of record fields is determined by their offsets (*offs*) relative to the record address. This offset constant is used in expression trees representing field selections (cf. Chapter 3, Figure 3.2). Procedure objects contain a procedure address (*adr*) as well as a reference to a statement sequence, i.e. a syntax tree representing the *body* of the procedure (cf. Section 5.2.3). The scope associated with a procedure can be retrieved from the procedure's type (cf. Section 5.2.2). Finally, predefined procedures and functions generating in-line code are represented as *StdProc* objects which are held in the *universe* scope. For *Module* objects, the reader is referred to Section 5.2.4.

It is convenient to represent also expressions as (anonymous) objects. Using an *Expr* node, all kinds of expressions can be represented in form of expression trees. Since expressions are extensions of ordinary objects, they can also be used as addresses for variables, as indicated in Chapter 3. Note that such expression objects must not be confused with *items* used in the back-end: whereas a single object statically represents an object (or expression) of a program, an item may represent several different objects during its lifetime (see also Section 5.3.1).

```
CONST
  not = 1; abs = 2; ord = 3; ... (* unary operations f *)
  add = 20; sub = 21; mul = 22; ... (* binary operations f *)

TYPE
  Expr = RECORD (Object)
    f: INTEGER;
    x, y: ↑Object
  END;

PROCEDURE Deref (VAR x: ↑Object);
PROCEDURE Index (VAR x: ↑Object; index: ↑Object);
PROCEDURE Select (VAR x: ↑Object; VAR fieldName: ARRAY OF CHAR);

PROCEDURE Not (VAR x: ↑Object);
PROCEDURE BinOp (VAR x: ↑Object; y: ↑Object; f: INTEGER);
PROCEDURE ElemTest (VAR x: ↑Object; set: ↑Object);
PROCEDURE TypeTest (VAR x: ↑Object; type: ↑Object; guard: BOOLEAN);

PROCEDURE StdFuncCall
  (VAR x: ↑Object; proc: ↑StdProc; VAR args: ARRAY OF ↑Object; nofArgs: INTEGER);
```

Expression trees are constructed using a small set of procedures, which usually accept one or two argument expression trees *x* and *y* and then return a new expression tree in *x*. Selectors are handled with the procedures *Deref*, *Index*, *Select* and *TypeTest* (with *guard* = TRUE). These four procedures perform the necessary type checks and conversions and generate expression trees as explained in Section 3.3. Expression trees for binary operations are constructed using the *BinOp*, *ElemTest* and *TypeTest* procedures. *Not* is used for the unary operation ~. Other unary operations such as negation are regarded as binary operations with neutral elements (e.g. –x = 0 – x). Calls of predefined functions are handled by the *StdFuncCall* procedure, which also performs the necessary checks. Note that no special object extension is used to represent predefined functions but they are considered as (named) unary and binary operations. As an example for the recursive construction of expression trees, the OVP procedure parsing an Oberon-V *Expression* is showed (cf. Appendix A.10):

```
PROCEDURE SimpleExpression (VAR x: ↑OVT.Object); ...

PROCEDURE Expression (VAR x: ↑OVT.Object);
   VAR y: ↑OVT.Object; sym: INTEGER;
BEGIN SimpleExpression(x);
   IF (OVS.eql <= OVS.sym) & (OVS.sym <= OVS.is) THEN sym := OVS.sym;
      OVS.Scan; SimpleExpression(y);
      CASE sym
      OF OVS.eql, OVS.lss THEN OVT.BinOp(x, y, sym)
      OF OVS.neq THEN OVT.BinOp(x, y, OVS.eql); OVT.Not(x)
      OF OVS.leq THEN OVT.BinOp(y, x, OVS.lss); OVT.Not(y); x := y
      OF OVS.gtr THEN OVT.BinOp(y, x, OVS.lss); x := y
      OF OVS.geq THEN OVT.BinOp(x, y, OVS.lss); OVT.Not(x)
      OF OVS.in THEN OVT.ElemTest(x, y)
      OF OVS.is THEN OVT.TypeTest(x, y, FALSE)
      END
   END
END Expression;
```

Objects associated with identifier nodes and objects used in expression trees may be *shared*. Thus, after declaring a variable x, i.e. after introducing an identifier node for x and associating a variable object with it, *this* object is used within an expression tree whenever the variable x occurs within an expression (Figure 5.2). It might be noteworthy, that in the comparable Oberon compiler OP2 [Crelier 1990], expression trees are represented by special nodes and for each object referred to within an expression tree, an additional node referring to this object is required. Since variables (and constants, etc.) are almost always leaf nodes (cf. Section 3.3, Figure 3.1) and since the number of leaf nodes is approximately one half of the number of all nodes within a binary expression tree, object sharing means a significant saving of memory space.

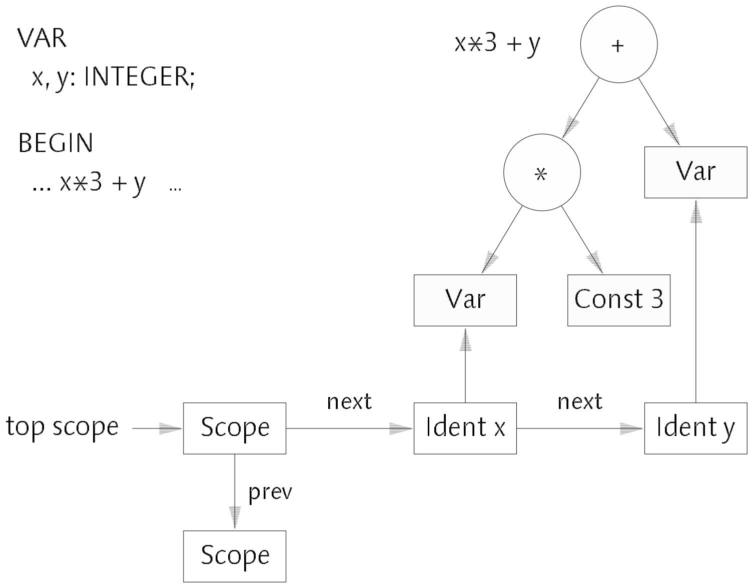


Figure 5.2 Sharing of Objects by Symbol Table and Expression Tree

Ranges are also extensions of *Objects*. As explained in Chapter 3, each range object contains the coefficients (*coeff*) of the affine function it represents. The lengths of all ranges introduced by a range declaration (i.e. the length of the range declaration) are held in a *Length* record. It is convenient to have a reference (*length*) from each range object to this length record. Since range objects may occur as indices, they may be subject to index constraints. Thus, to each range object, a list of *Constraint* nodes (*cnstr*) may be associated. Each constraint is clearly determined by a *range* and the *length* of the subscripted array (cf. Section 3.9). For a snapshot of the symbol table state after processing a range declaration, the reader is referred to Figure 3.4 in Chapter 3.

```
CONST
   MaxNofDims = 4;

TYPE
   Constraint = RECORD
      next: ↑Constraint;
      range: ↑Range;
      length: ↑Object
   END;

   Length = RECORD
      length: ARRAY MaxNofDims+1 OF ↑Object;
      nofDims: INTEGER
   END;
```

```
    Range = RECORD (Object)
        coeff: ARRAY MaxNofDims+1 OF ↑Object;
        cnstr: ↑Constraint;
        length: ↑Length
    END;

PROCEDURE RangeLen (VAR len: ↑Object; lo, hi: ↑Object);
PROCEDURE NewRange (VAR ran: ↑Object; c0, ci: ↑Object; i: INTEGER; l: ↑Length);
```

The length of a (1-dimensional) range is computed with the procedure *RangeLen* using the lower and upper bound of the range. RangeLen performs also a minor optimization: For the frequent case of ranges of the form a .. b–1, an expression tree corresponding to the expression b–a (or b, if a = 0) instead of ((b–1) – a) + 1 is returned. A new range object is created with *NewRange* which uses the coefficients c_0 and c_i as well as a reference to the length (*l*) of the range declaration (cf. Section 3.4).
As explained in Section 3.5, ranges are treated in a special way when they are involved in linear expressions, i.e. as operands of the operations +, –, and ∗. For each of these operations special procedures *Add*, *Sub* and *Mul* exist in OVT which perform the necessary transformations. Note that these (internal) procedures are used by the *BinOp* procedure as well as the procedures *Deref*, *Index* and *Select* (since also address computations are treated uniformly). Indeed, almost all additional effort required to handle range expressions is covered by these three procedures. Below the full procedure *Add* is shown without further explanations. The procedures for – and ∗ are quite similar (*NewInt*, *NewExpr*, *Clone* and *CollectConstraints* are frequently used auxiliary procedures).

```
PROCEDURE NewInt (VAR x: ↑Object; typ: ↑Struct; int: INTEGER);
    VAR c: ↑Const;
BEGIN NEW(c); c.typ := typ; c.int := int; x := c
END NewInt;

PROCEDURE NewExpr (VAR x: ↑Object; y: ↑Object; typ: ↑Struct; f: INTEGER);
    VAR e: ↑Expr;
BEGIN NEW(e); e.typ := typ; e.f := f; e.x := x; e.y := y; x := e
END NewExpr;

PROCEDURE Clone (VAR r: ↑Range; from: ↑Range);
BEGIN NEW(r); r.typ := from.typ;
    ALL i = 0 .. MaxNofDims DO r.coeff[i] := from.coeff[i] END;
    r.cnstr := from.cnstr; r.length := from.length
END Clone;
```

```
PROCEDURE CollectConstraints (VAR c: ↑Constraint; d: ↑Constraint);
  VAR q: ↑Constraint;
BEGIN
  WHILE d # NIL DO
    NEW(q); q.next := c; c := q;
    q.range := d.range; q.length := d.length;
    d := d.next
  END
END CollectConstraints;

PROCEDURE Add (VAR x: ↑Object; y: ↑Object);
  VAR h: ↑Object; r, ry: ↑Range; i: INTEGER;
BEGIN
  ASSERT((x.typ = y.typ) & ((x.typ = intTyp) | (x.typ = adrTyp)));
  IF x IS Const THEN h := x; x := y; y := h END;
  IF x IS Const THEN NewInt(x, x.typ, x{Const}.int + y{Const}.int)
  ELSIF ~(y IS Const) | (y{Const}.int # 0) THEN
    IF x IS Range THEN h := x; x := y; y := h END;
    IF x IS Range THEN
      Clone(r, x{Range}); ry := y{Range}; i := ry.length.nofDims;
      WHILE i >= 0 DO Add(r.coeff[i], ry.coeff[i]); DEC(i) END;
      CollectConstraints(r.cnstr, ry.cnstr); x := r
    ELSIF y IS Range THEN Clone(r, y{Range}); Add(x, r.coeff[0])
    ELSE NewExpr(x, y, x.typ, add)
    END
  END
END Add;
```

Last but not least, two additional extensions of Objects are required in OVT: *FuncCall* and *Address* objects. While the former are used to represent function calls within expression trees (cf. Section 5.2.3), the latter represent machine specific addresses and hence provide an interface to the compiler back-end. Address objects usually are extended in a back-end by machine specific attributes (e.g. register numbers or memory addresses). They are created by a back-end procedure to be installed in the global variable *NewAddress*.

```
TYPE
  FuncCall = RECORD (Object)
    proc: ↑Object;
    args: ↑Assignment
  END;

  Address = RECORD (Object)
    level: INTEGER;
    leaf: BOOLEAN
  END
```

```
VAR
   NewAddress: PROCEDURE (level: INTEGER): Address;
```

The scope level of an address is stored in its *level* field; the level zero indicates an address of a global variable (or procedure). If an address is used within an address computation or if it is referred to from within a nested scope, the *leaf* field of the address object is set to FALSE. This information may be used by a back-end to decide whether a variable might be held in a register or whether it must be held in memory.

5.2.2 Representation of Types

Each object is associated with a *Struct* node which describes the structure of the object's type. Like objects, different structures are best represented by type extensions of the base type *Struct*. If a type has a name, the corresponding Struct node contains a reference to the associated identifier (*id*). The *size* of a type is the number of storage units required for variables of this type. This value is inherently machine specific and hence is computed by means of the procedure *SetSize* installed by the back-end. Arrays and records are structured types and thus represented by extensions of the type *Structured*. This additional extension level allows for convenient tests in the compiler (e.g. "IF typ IS Structured THEN ...") and for more expressiveness (e.g. a pointer's base type must always be structured). Structured types also may have a type descriptor's address (*desc*) associated with them which is used to implement type guards and type tests (cf. [Cohen 1991]).

```
TYPE
   Struct = RECORD (Struct)
      size: ↑Const;
      id: ↑Object
   END;

   Structured = RECORD (Struct)
      desc: ↑Address
   END;

   Array = RECORD (Structured)
      length: ↑Const; (* length = NIL <=> open array *)
      elemTyp: ↑Struct
   END;
```

```
    Record = RECORD (Structured)
      baseTyp: ↑Record;
      scope: ↑Scope
    END;

    Pointer = RECORD (Struct)
      pointeeTyp: ↑Structured
    END;

    Procedure = RECORD (Struct)
      scope: ↑Scope;
      resTyp: ↑Struct (* resTyp # NIL <=> function *)
    END;

  VAR
    SetSize: PROCEDURE (typ: ↑Struct);
```

Arrays are determined by their *length* and the *element type*. Records are described by their *base type* and their record fields, which are collected within the record *scope*. If a record type R1 is an extension of a record R, the *prev* field of the scope record associated with R1 points to the scope record associated with R (Figure 5.3). Thus, also record fields can be retrieved using the *This* function mentioned in the beginning.

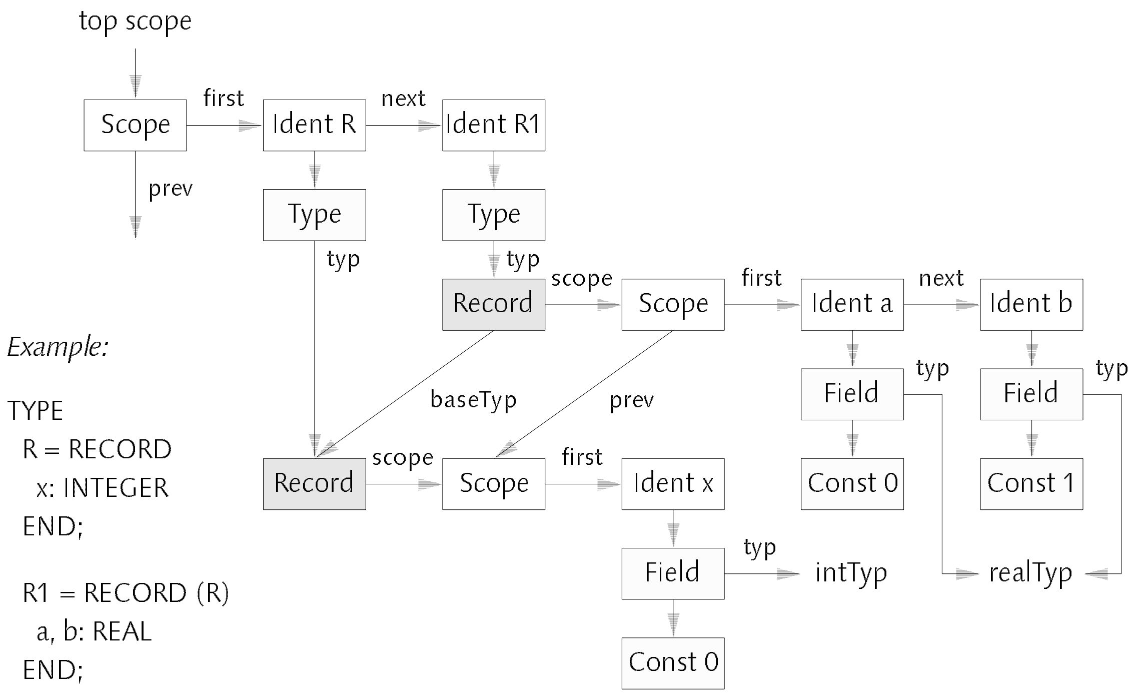
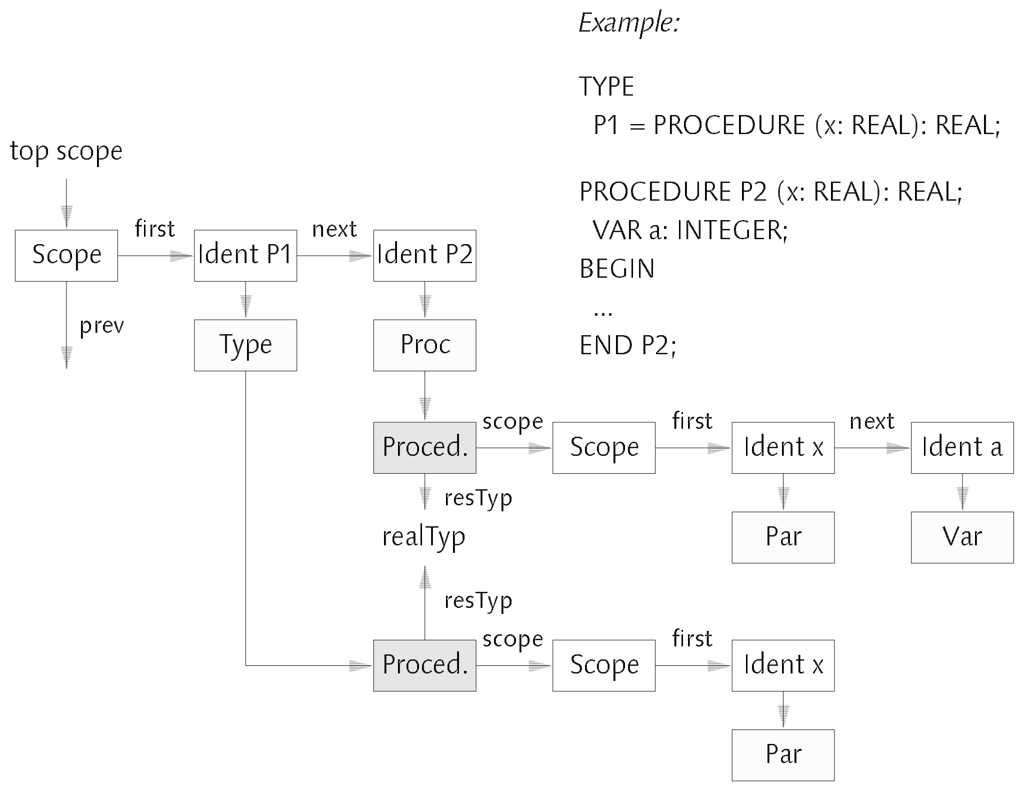


Figure 5.3 Representation of Records

All information used for a pointer type is its base type (*pointeeTyp*). Procedure types are determined by the procedure's parameter list and their result type, if any. The procedure parameters as well as local variables are held in the same scope, the parameters beeing distinguished from local variables by the type of the representing object (*Par* instead of *Var*). Furthermore, parameters, if any, are always at the beginning of an identifier list. The types of procedure objects (*Proc*) and the structure belonging to procedure types are represented in the same way, namely by Struct nodes associated with a scope containing the procedure parameters (Figure 5.4).



Example:

```
TYPE
  P1 = PROCEDURE (x: REAL): REAL;

PROCEDURE P2 (x: REAL): REAL;
  VAR a: INTEGER;
BEGIN
  ...
END P2;
```

Figure 5.4 Representation of Procedures and Procedure Types

Finally, the module OVT provides a set of initialized global *Struct* variables which provide references to the Oberon-V *basic types* (cf. Appendix A.6.1) and a few internally used types (such as e.g. *nilTyp* or *stringTyp*):

```
VAR
  undefTyp, adrTyp, nilTyp, stringTyp,
  boolTyp, charTyp, setTyp, intTyp, realTyp, cmplxTyp: ↑Struct;
```

The type *adrTyp* deserves a special explanation: Because the intermediate representation of expressions does *not* distinguish between computations explicitly specified in a program and address computations implicitly specified via selectors, it is impossible to decide afterwards, which expressions represent "normal" computations and which represent implicit address computations (remember that the unified representation of expressions is *required* to handle range expressions). However, a back-end may generate significantly better code, if it is known which computations can be performed in address registers (if there are any) and which computations must be performed in other (scalar) registers. Furthermore, for address computations, explicit addition of constant values may be deferred and sometimes even is not necessary (e.g. if the address to be computed is used for a load instruction that allows for immediate operands). For normal computations such an approach would change the semantics of the computation. In OVT, all operands of address computations are first converted into "addresses", i.e. a conversion node is introduced if necessary. Figure 5.5 illustrates the use of an address conversion node within an index address computation. A particular back-end may interpret such a conversion node such that code is generated to move the operand into an address register. Note that only integer values are converted into addresses. Like "linear operations" (cf. Section 3.5), a conversion applied to a range object is transformed to conversions applied to the ranges' coefficients.

Example (without index checks):

```
VAR
  A: ARRAY 100 OF REAL;

BEGIN
  ... A[i] ...
```

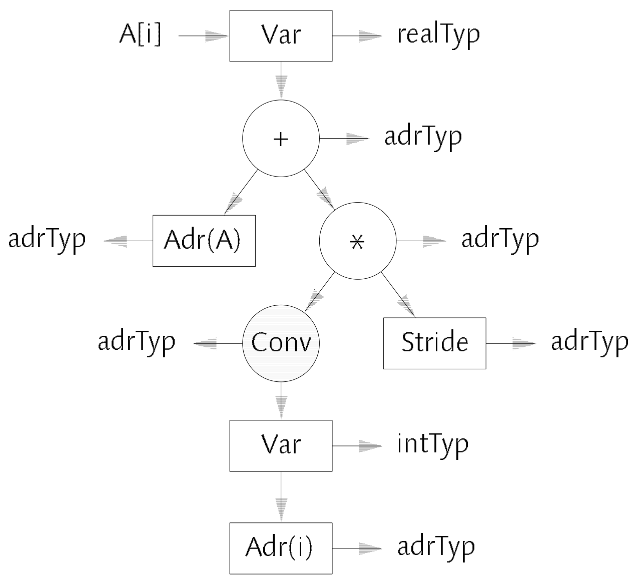
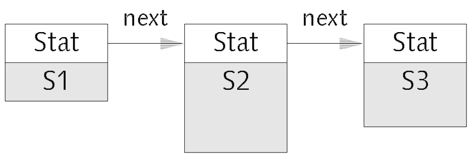


Figure 5.5 Conversion of an Integer Index into an Address

5.2.3 Representation of Statements

Procedures and modules may have *bodies* containing *statement sequences*. Statement sequences are represented as lists of extended *Stat* records (Figure 5.6). For each statement, an appropriate record extension is used.

```
TYPE
  Stat = RECORD
    next: ↑Stat
  END;
```

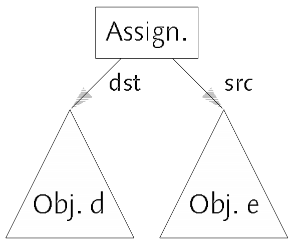


S1; S2; S3

Figure 5.6 Representation of Statement Sequences

Assignments are represented by two expression trees, one for the left-hand side designator and one for the right-hand side expression of the assignment (Figure 5.7). An new assignment statement is generated using the procedure *Assign* which checks whether the assignment is legal and which introduces additional *conversion nodes* if necessary (e.g. to convert the result of an integer expression to a real value). The same procedure is also used to generate parameter lists for procedure calls (with *param* = TRUE, see below):

```
TYPE
  Assignment = RECORD (Stat)
    dst, src: ↑Object
  END;
```



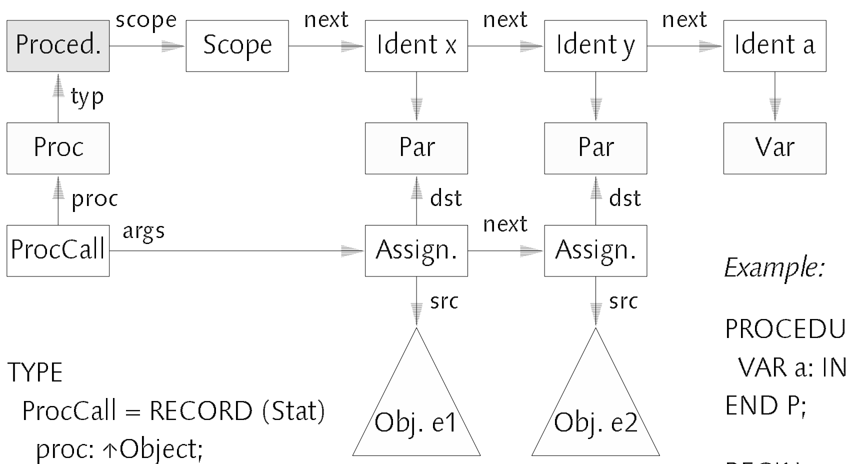
Example:

d := e

```
PROCEDURE Assign (VAR stat: ↑Stat; dst, src: ↑Object; param: BOOLEAN);
```

Figure 5.7 Representation of Assignments

Procedure calls are represented by *ProcCall* nodes, which contain a reference to the called procedure object (which may be a procedure variable, too) as well as a list of assignments. Each assignment corresponds to a pair (parameter, argument). Figure 5.8 illustrates the data structure corresponding to a call of a procedure P. The parameter objects are found in the scope of the called procedure.



```
TYPE
  ProcCall = RECORD (Stat)
    proc: ↑Object;
    args: ↑Assignment
  END;
```

Example:

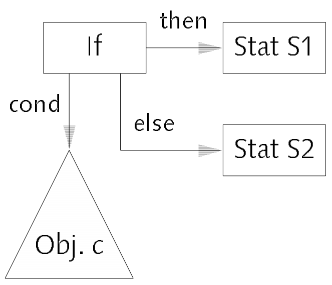
```
PROCEDURE P (x, y: INTEGER);
  VAR a: INTEGER;   ...
END P;

BEGIN
   P(e1, e2); ...
```

Figure 5.8 Representation of Procedure Calls

The IF statement is represented using an *If* node. Because ELSIF branches can be considered as nested IF statements (cf. Appendix A.11.5), no special ELSIF node is required (Figure 5.9). Nevertheless, the same code quality can be achieved with a straight-forward translation.

```
TYPE
  If = RECORD (Stat)
    cond: ↑Object;
    then, else: ↑Stat
  END;
```



Example:

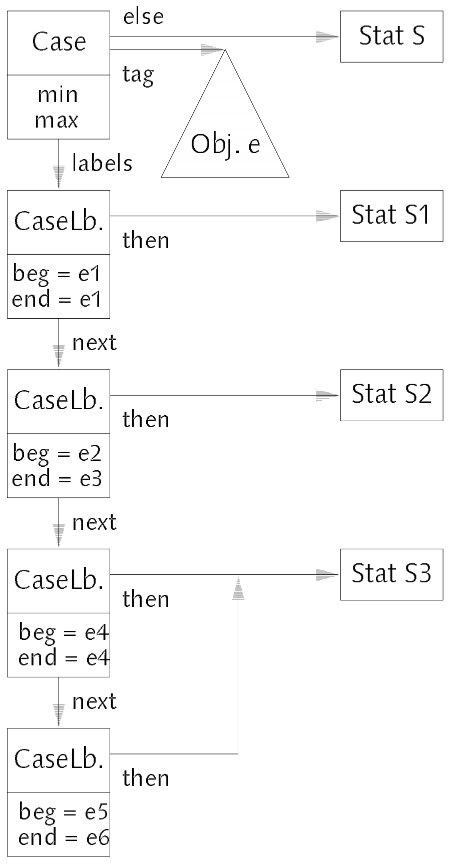
```
IF c THEN S1
ELSE S2
END
```

Figure 5.9 Representation of IF Statements

Syntactically, the CASE statement is the most complicated statement and thus requires a more complex data structure for its representation than other statements. The *Case* node contains a reference to the *tag* expression, a pointer to a list of case labels as well as the minimum and maximum value assumed by these case labels. Case labels which belong to the same case label list of a CASE statement follow each other in the *labels* list and point to the same statement sequence (*then*). Each *CaseLabels* entry is determined by two values *beg* and *end*, which denote the case labels in the set {beg, beg+1, ... end}. If the set contains only a single element only, then beg = end (Figure 5.10).

```
TYPE
  Case = RECORD (Stat)
    tag:    ↑Object;
    min, max: INTEGER;
    labels: ↑CaseLabels;
    else: ↑Stat
  END;

  CaseLabels = RECORD
    next: ↑CaseLabels;
    beg, end: INTEGER;
    then: ↑Stat
  END;
```



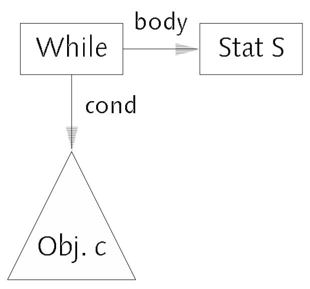
Example:

```
CASE e
OF e1 THEN S1
OF e2 .. e3 THEN S2
OF e4, e5 .. e6 THEN S3
ELSE S
END
```

Figure 5.10 Representation of CASE Statements

The REPEAT and WHILE statements are very similar; Figures 5.11 and 5.12 show the corresponding data definitions.

```
TYPE
  While = RECORD (Stat)
      cond: ↑Object;
    body: ↑Stat
  END;
```

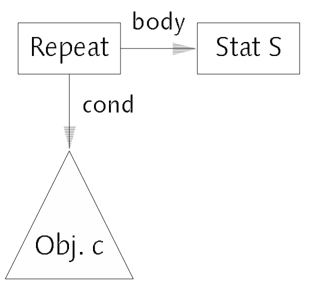


Example:

```
WHILE c DO S END
```

Figure 5.11 Representation of WHILE Statements

```
TYPE
  Repeat = RECORD (Stat)
      cond: ↑Object;
    body: ↑Stat
  END;
```



Example:

REPEAT S UNTIL c

Figure 5.12 Representation of REPEAT Statements

The *Return* record type contains a reference to the returned object (*val*), if any. A special *Stat* extension called *BuiltIn* is used to represent calls of predefined procedures such as NEW or HALT; such procedures are identified by a procedure number (*f*) and they may have at most three arguments *x*, *y*, and *z*. BuiltIn statement nodes are created using the *StdProcCall* procedure, which also performs the necessary checks.

```
TYPE
  Return = RECORD (Stat)
    val: ↑Object (* val = NIL <=>  no return expression *)
  END;

  BuiltIn = RECORD (Stat)
    f: INTEGER;
    x, y, z: ↑Object
  END;

PROCEDURE StdProcCall
  (VAR stat: ↑Stat; proc: ↑StdProc; VAR args: ARRAY OF ↑Object; nofArgs: INTEGER);
```

Finally, a representation for ALL statements is required. As explained in Chapter 3 (Section 3.7), an n-dimensional ALL statement is transformed into n nested loops with corresponding *prolog* and *epilog* assignment sequences. Each loop needs a *counter* variable and an initial value for this counter, namely the *length* of the range declaration in this dimension. If the length of the innermost loop is short, i.e. if the length of the n-th loop is constant and less than VL_{max}, no innermost loop is required at all. This is signaled by a missing counter variable for this loop (i.e. by counter[nofDims] = NIL). Furthermore, a list of constraints (*cnstr*) to be checked before executing the ALL statement is provided (cf. Section 3.9, Index Checks). The *body* field in the *All* record points to the statement sequence corresponding to the transformed assignment sequence enclosed by the ALL statement.

```
TYPE
  All = RECORD (Stat)
    prolog, epilog: ARRAY MaxNofDims+1 OF ↑Stat;
    length: ARRAY MaxNofDims+1 OF ↑Object;
    counter: ARRAY MaxNofDims+1 OF ↑Var;
    nofDims: INTEGER;
    cnstr: ↑Constraint;
    body: ↑Stat
  END;

PROCEDURE AllStat
  (VAR stat: ↑Stat; lengths: ARRAY OF ↑Object; nofDims: INTEGER; scope: ↑Scope);
```

After compilation of the range declaration and the assignment sequence of an n-dimensional ALL statement, the assignment sequence (*stat*), the *length* of the range declaration, the number of dimensions (*nofDims*) as well as the current *scope* are passed as arguments to *AllStat*. AllStat transforms the assignment sequence as described in Section 3.7.2 and creates the necessary prolog and epilog statements. Auxiliary variables introduced during the transformation as well as counter variables are simply appended in the current scope, using identifier names which cannot be used in a legal Oberon-V program. Thus, they are invisible to the programmer but can be handled like other variables by the compiler.

5.2.4 Representation of Entire Programs

Eventually one may have a look at the representation of entire programs. In Oberon-V a compilation unit is a module. Modules are represented by *Module* objects:

```
Type
  Module = RECORD (Proc)
    fingerprint: INTEGER
  END;
```

From an operational view point, modules can be regarded as special procedures, and therefore they are represented by extensions of *Proc* objects (it is also convenient to do so, since code generation for modules and for procedures is very much the same. Thus, this extension relation allows for a similar treatment of procedures and modules in the back-end). The *fingerprint* field is used for interface checks and is computed of the symbol file (see Section 5.2.5). As with procedure objects, a module scope and a statement sequence are associated with a module object. The module scope contains all declarations, i.e. all globally declared identifiers which are used to refer to any (globally visible) object in the module. Note that also *imported* modules are contained within this scope. Since local scopes and statement sequences of procedures are associated with the corresponding procedure objects, which again are referred to by its identifiers, a module identifier associated with a module object may represent an entire program. This fact is also reflected by the interface of the parser OVP (*mid* is the module identifier corresponding to the compiled module):

```
MODULE OVP (OVT);
  PROCEDURE Module (VAR mid: ↑OVT.Ident);
END OVP.
```

5.2.5 *Import and Export*

In an Oberon-V module, globally declared objects can be made available to other modules by means of *exporting* the corresponding identifiers. Vice versa, exported objects of other modules can be used by means of *importing* the corresponding identifiers of these modules (cf. Appendix A, Sections A.4 and A.12). The collection of all exported objects constitutes a module's *interface*. Similar import/export mechanisms can be found in Modula-2 [Wirth 1885] and Oberon [Wirth 1988a].

In order to implement import and export safely, a compiler must know the interfaces of imported modules. Usually this information is contained in so-called *symbol files*, which are generated as by-products of module compilations and which are read during the compilation of import lists. Roughly speaking, a symbol file contains a linearized description of a subset of the symbol table, namely the subset of all exported identifiers together with their associated object and type information. Thus, generating a symbol file means linearizing the symbol table, and reading a symbol file means reconstructing parts of the symbol table from its linearized description. Although basic algorithms for the linearization of graphs and their reconstruction are not difficult, their application to symbol files is significantly more subtle, mostly because a *single* type can be imported *several times* through different modules. Thus, it is necessary to recognize whether two types imported from different modules actually denote the same type or not. As a further requirement, it is necessary that already compiled modules can easily detect whether their imported modules have changed their interfaces. A simple solution for this problem is to compute a (statistically) unique *fingerprint*, e.g.

an integer, from a symbol file and to use this fingerprint as an identification for the whole interface. If the interface changes, also the fingerprint changes.

A new method for import and export based on a linearization algorithm for general graphs has been developped. Since the problem is not directly related to the topic of this thesis, its implementation is not described here, but the interested reader is referred to [Griesemer 1991]. The main advantage of this algorithm when compared with a solution currently used in Oberon compilers [Gutknecht 1985] is that *recursive type definitions* do not impose any problems at all, whereas Gutknecht's solution requires a special treatment for recursive type definitions via pointers (and would also for other forms of recursive type definitions, if they were allowed in current Oberon compilers). An elaborate treatment of the import/export topic will be found in [Crelier 1993]. Also a mechanism to extend a module interface afterwards without changing its fingerprint will be described there.

5.3 Code Generation

After a compilation unit has been successfully parsed and the intermediate data structure has been built, this data structure is passed to the back-end. The back-end consists of four modules: The high-level code generator *OVC*, the module *OVE* generating code patterns for expressions, the module OVV which manages the virtual code generated by *OVE*, and the low-level module *OVO* which assembles the final Cray object file. In the following a rough survey of the modules OVC, OVE and OVV is given.

5.3.1 OVC: High Level Code Generation

The module OVC provides three services for code generation, namely the procedures *SetSize*, *AllocSpace* and *GenCode* and thus corresponds to a similar module called *OPV* of the Oberon-2 compiler OP2 [Crelier 1990]:

```
MODULE OVC (OVT);
   PROCEDURE SetSize (typ: ↑OVT.Struct);
   PROCEDURE AllocSpace (pid: ↑OVT.Ident; VAR GSize: INTEGER);
   PROCEDURE GenCode (pid: ↑OVT.Ident);
END OVC.
```

SetSize computes the size required for variables of arbitrary type (*typ*); the procedure is installed in the global variable OVT.SetSize (cf. Section 5.2.2) by the command module OV. Before starting code generation via GenCode, the procedure AllocSpace is called which computes the addresses of variables (and parameters) local to a procedure or module specified by their identifiers (*pid*). For each scope, all identifiers are sorted according to the value of their *usage* field (cf. Section 5.2.1). Variables referred to by identifiers with high usage are processed first and hence register locations may be allocated for them, if their addresses are *leaf*. The address information for variables as well as additional data (such as the total size of all local variables, etc.) are held in extensions of *Address* objects associated with *Var* and *Par* (or *Proc*) objects; the extension is defined in module OVE. The global data size (*GSize*) is also computed by AllocSpace. Note that no distinction is required between variables in local scopes and variables in module scopes, but address allocation is determined only by the *exported* and *usage* information of Ident nodes as well as the *level* and *leaf* fields of Address objects (cf. Section 5.2.1). Thus, global variables that are used only in the initialization statement sequence of a module may also be allocated in registers or on the run-time stack. Finally, GenCode is used to traverse the scope of a procedure or module (*pid*). GenCode is called recursively for each local procedure. After traversing a procedure's scope, GenCode generates code for the syntax tree corresponding to the procedure's statement sequence.

Code generation can be regarded as a transformation of the syntax tree into a linearized sequence of machine instructions. A systematic and efficient approach is to map every node of the syntax tree into a semantically equivalent code sequence, which then is appended to the code already generated. In the Oberon-V compiler, a simple technique is used which proved to be successful in a series of similar compilers [Wirth 1985a][Wirth 1992][Brandis et. al. 1992]. In order to generate code for a node S, first, code is generated for all its descendant nodes S_1, S_2, ... S_n. Information about the code generated for a node S_k is returned in an *attribute* x_k. The attributes x_1, x_2, ... x_n are then used to generate the code for S. The attribute x describing the code for S may in turn be used by the parent of S, and so on. Thus, if the (well-understood) influence of the symbol table is ignored, a context-free translation of nodes S_k is obtained. For a given node S, its code Code(S) is a function F_S of the Code(S_k) generated for its descendant nodes S_k:

$$\text{Code}(S) = F_S(\text{Code}(S_1), \text{Code}(S_2), \ldots \text{Code}(S_n))$$

This translation scheme yields an *attribute flow* along the edges of the syntax tree during its traversal (Figure 5.13). The attribute x_k typically represents information about the result of a computation S_k, e.g. a register number, a memory address or a condition code. Attributes are usually represented by records called *Items*. Since the attribute flow corresponds to the call order of the procedure traversing the syntax tree, items simply can be passed as arguments to this procedure.

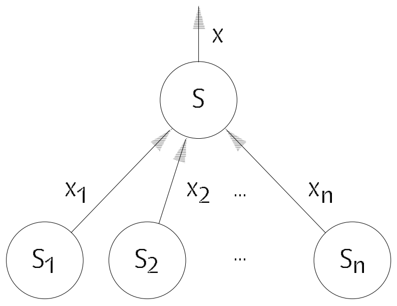


Figure 5.13 Attribute Flow during Code Generation

In order to give an idea of how this tree traversal proceeds, parts of the *CExpression* procedure for conditional expressions are shown below. The procedure recursively traverses an expression tree (*obj*) and yields an item (*x*). After traversing the descendants of an *Expr* node (cf. Section 5.2.1), code is generated for this node by means of procedures from module OVE; these procedures correspond to the functions F_S. The boolean operations *and* and *or* (& and |) do not only compute a *value* but also change the *control flow* of the generated code (e.g. the second operand of an *or* operation must be evaluated only if the first operand evaluated to FALSE). This *conditional evaluation* requires additional information in order to generate the necessary jumps. Without going too much into details, it is mentioned that these information is provided by means of two *label* arguments, namely the label *tjmp* denoting the jump destination in case of an evaluation to TRUE and *fjmp* denoting the jump destination in case of an evaluation to FALSE (see also [Wirth 1992]) These labels are also used to generate code for conditional statements depending on the value of their corresponding conditional expression (note that *setting a label* with OVV.Label may involve a list of forward jumps to be fixed up). The procedure *Expression* is used when no conditional result is expected.

```
PROCEDURE Expression (VAR x: OVE.Item; obj: ↑OVT.Object);
   VAR tjmp, fjmp: INTEGER;
BEGIN tjmp := OVV.NewLabel; fjmp := OVV.NewLabel;
   CExpression(x, obj, tjmp, fjmp); OVE.Value(x, tjmp, fjmp)
END Expression;
```

```
PROCEDURE CExpression
   (VAR x: OVE.Item; obj: ↑OVT.Object; VAR tjmp, fjmp: INTEGER);
   VAR y: OVE.Item; ran: ↑OVT.Range; expr: ↑OVT.Expr; L: INTEGER;
BEGIN
   IF obj IS OVT.Address THEN OVE.GetAddress(x, obj{OVE.Address})
   ELSIF obj IS OVT.Const THEN OVE.GetConst(x, obj{OVT.Const})
   ELSIF obj IS OVT.Var THEN Expression(x, obj{OVT.Var}.adr); OVE.GetVar(x)
   ELSIF obj IS OVT.Proc THEN OVE.GetAddress(x, obj{OVT.Proc}.adr{OVE.Address})
   ELSIF obj IS OVT.Range THEN ran := obj{OVT.Range};
      Expression(x, ran.coeff[0]); Expression(y, ran.coeff[1]); OVE.GetVector(x, y)
   ELSIF obj IS OVT.Expr THEN expr := obj{OVT.Expr};
      IF expr.f = OVT.and THEN L := OVV.NewLabel;
         CExpression(x, expr.x, L, fjmp); OVE.Condition(x); OVE.Not(x);
         OVE.Jump(x, fjmp);
         OVV.Label(L); CExpression(x, expr.y, tjmp, fjmp); OVE.Condition(x)
      ELSIF expr.f = OVT.or THEN L := OVV.NewLabel;
         CExpression(x, expr.x, tjmp, L); OVE.Condition(x); OVE.Jump(x, tjmp);
         OVV.Label(L); CExpression(x, expr.y, tjmp, fjmp); OVE.Condition(x)
      ELSIF expr.f = OVT.not THEN CExpression(x, expr.x, fjmp, tjmp); OVE.Not(x)
      ELSIF expr.y = NIL THEN Expression(x, expr.x); (* unary operation *)
         CASE expr.f
         OF OVT.abs THEN OVE.Abs(x)
         OF OVT.ord THEN OVE.Ord(x)
         ...
         END
      ELSE Expression(x, expr.x); Expression(y, expr.y); (* binary operation *)
         CASE expr.f OF
         OF OVT.add THEN OVE.Add(x, y)
         OF OVT.sub THEN OVE.Sub(x, y)
         ...
         END
      END
   ELSIF obj IS OVT.FuncCall THEN ...
   END;
   x.typ := obj.typ
END CExpression;
```

5.3.2 OVE: Expressions

The module OVE could be titled the "work horse" of the back-end modules, since its procedures effectively translate an abstract operation such as an addition or an assignment into machine instructions. However, in order to simplify code generation and to enable some optimizations, OVE does not generate Cray code but code for a *virtual machine*. The difference between this *virtual code* and Cray code is, that an *unbounded* number of registers for expression evaluation is assumed. The instructions themselves are (nearly) the same. As will be explained in the next section, this allows for some powerful optimizations. Furthermore, it relieves module OVE from handling register allocation for temporary values (i.e. for expression results, etc.).

To provide the prerequisites for understanding the OVE data structures, a brief description of the Cray architecture is given. A Cray Y-MP processor offers 5 different register files, namely 8 scalar address registers (*A-registers*) of 32 bits width, 8 scalar general-purpose registers (*S-registers*) of 64 bits width and 8 vector registers (*V-registers*). Each vector register may hold at most 64 elements of 64 bits width each, i.e. $VL_{max} = 64$. Additionally, there are two register files with 64 *temporary registers* each, called *B-* and *T-registers*. B-registers have a width of 32 bits, T-registers may hold 64-bit values. These temporary registers are used to hold local variables and to pass procedure arguments. Address computations are performed using the A-registers; conventional scalar arithmetics is done using the S-registers and vector instructions operate on the V-registers. The instruction set is very similar to the instruction set of the hypothetical machine introduced in Section 1.1. Note that for almost all operations *op* three instruction variants exist:

Si := Sj *op* Sk
Si := Sj *op* Vk
Si := Vj *op* Vk

For a general description of the Cray architecture and its instruction set, the reader is referred to [Russel 1978], [Robbins 1987] and [Cray 1988]. In the following, only a rough survey of the internals of the module OVE is given. The only data types exported are the type *Address*, an extension of OVT.Address, as well as the *Item* data type (where only the *typ* field is exported):

```
Address = RECORD (OVT.Address)
   mode, reg, adr, rel: INTEGER;
   LSize, ... : INTEGER
END

Item = RECORD
   typ: ↑OVT.Struct;
   mode, reg, ireg, adr, rel, n: INTEGER
END
```

The *mode*, *reg*, *adr* and *rel* fields of an Address object specify the location of a variable or procedure: the *mode* field determines, which of the remaining fields is valid; *reg* corresponds to a register number; *adr* and *rel* specify a memory address and possible relocation information required to generate Cray object files. The fields *LSize*, etc. are used for procedures (and modules) only and specify the stack requirements and the number of local variables held in registers.

The architecture of a code generator and thus also the module OVE is strongly influenced by the addressing modes provided by the target machine. These addressing modes find their direct representation in the different *modes* of *items*. An *Item* record specifies (the location of) a value. Figure 5.14 shows the possible contents of item records and their interpretation. Note that using type extension to represent different item modes would be unpractical because items frequently change their modes during their life times, and every mode change would require to allocate a new item.

mode	*reg*	*ireg*	*n*	*adr*	*rel*	*meaning*
BregN ∗			x			B_n
TregN ∗			x			T_n
Mem ∗			level	x	x	$(FP_{level}) + adr_{rel}$
Proc ∗			level	label	x	adr_{rel}
Const			value			constant
Breg			x			(B_n)
Treg			x			(T_n)
Areg	x					(A_{reg})
AregO	x			x	x	$(A_{reg}) + adr_{rel}$
AregI	x			x	x	$M[(A_{reg}) + adr_{rel}]$
Sreg	x	(x)				(S_{reg})
Cond	x		cond			condition
Vreg	x	(x)				$(V_{reg}[i])$
VregO	x	x		x	x	$(A_{reg}) + (V_{ireg}[i]) + adr_{rel}$
VregI	x	x				$M[(A_{reg}) + (V_{ireg}[i])]$
Vector	x	stride				$(A/S_{reg}) + i*(A/S_{ireg})$
VectorI	x	stride				$M[(A_{reg}) + i*(A_{ireg})]$
VCond	x		cond			vector mask

Figure 5.14 Item Modes and Interpretation

Modes which are marked (∗) appear also as modes for address objects. A-, S- and V-register numbers (fields *reg* and *ireg*) specify virtual registers; they start with the number 8. Register numbers smaller than 8 are interpreted as immediate operands. Especially, the register number 0 denotes the value 0 (or 0.0, depending on the item type) and the register number 1 the value 1 (or 1.0

respectively). This encoding is useful to improve code generation for operations on complex numbers and also to denote special vector strides (e.g. vectors with stride 1; they can be loaded with a special instruction).

Constant values (mode = *Const*) are always held in an item record itself (field *n*). A value held in a register is represented by an item with mode *Areg*, *Sreg*, *Vreg*, *Breg* or *Treg*; the register number is specified by the *reg* field. In case of a complex number, i.e. where the item type is *cmplxTyp*, the *ireg* field specifies the S- or V- register holding the imaginary part of the number or vector (which may also be a constant, namely 0.0 or 1.0, using the encoding scheme explained before). Real and imaginary parts held in T-registers or in main memory are always stored consequtively, thus a single address suffices in these cases. Variables held in memory are represented by items with modes *AregI*, *Mem*, *VregI* and *VectorI*. The relocation field *rel* is valid only if it contains a positive value.

The *BregN*, *TregN*, *AregO*, *VregO* and *Vector* modes deserve a special explanation: Since no distinction is made between conventional and address computations by the front-end, and because addresses (which may be simple register *numbers*) appear also in expression trees, they consequently appear also as items. Here, these modes simply denote a B- or T-register *number*, a (scalar) *memory address*, a *vector of memory addresses* or a *vector address* respectively. During traversal of an expression tree, each occurrence of a *Var* node implies a corresponding transition of the corresponding item mode from a mode denoting an address to a mode denoting a variable at this address. The procedure *GetVar* performs this transition:

```
PROCEDURE GetVar (VAR x: Item);
   VAR v: INTEGER;
BEGIN
   CASE x.mode OF
   OF BregN THEN x.mode := Breg
   OF TregN THEN x.mode := Treg
   OF Breg, Treg, Areg, AregI, Sreg THEN LoadA(x); x.mode := AregI; x.adr := 0; x.rel := –1
   OF AregO THEN x.mode := AregI
   OF Vreg, VregI, VectorI THEN LoadV(x); x.mode := VregI; x.ireg := x.reg; x.reg := 0
   OF VregO THEN v := x.ireg; x.mode := AregO; ToAreg(x); x.mode := VregI; x.ireg := v
   OF Vector THEN x.mode := VectorI
   END
END GetVar;
```

Figure 5.15 illustrates the OVE procedure calls, the transition of the item modes and the code generated for a simple addition of two variables. The (small) numbers 1, 2, ... 5 indicate the visiting order of each node.

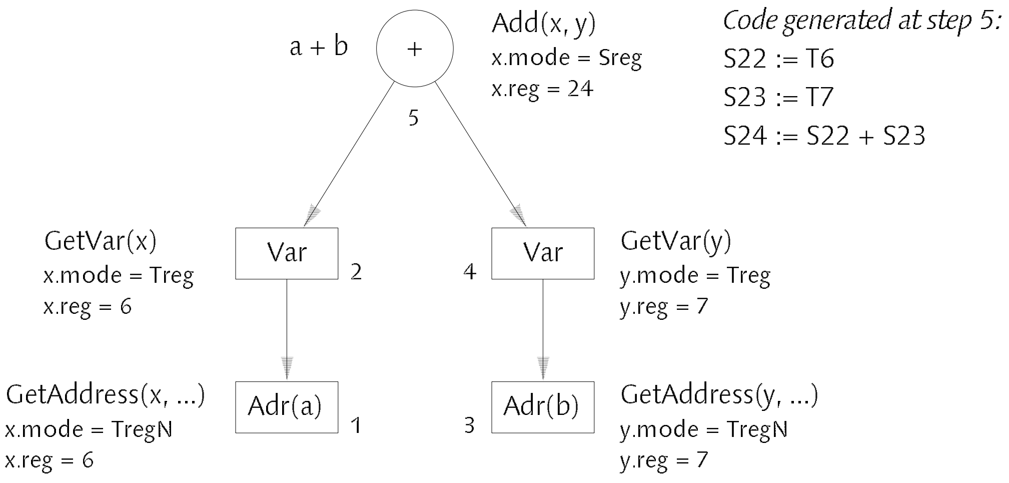
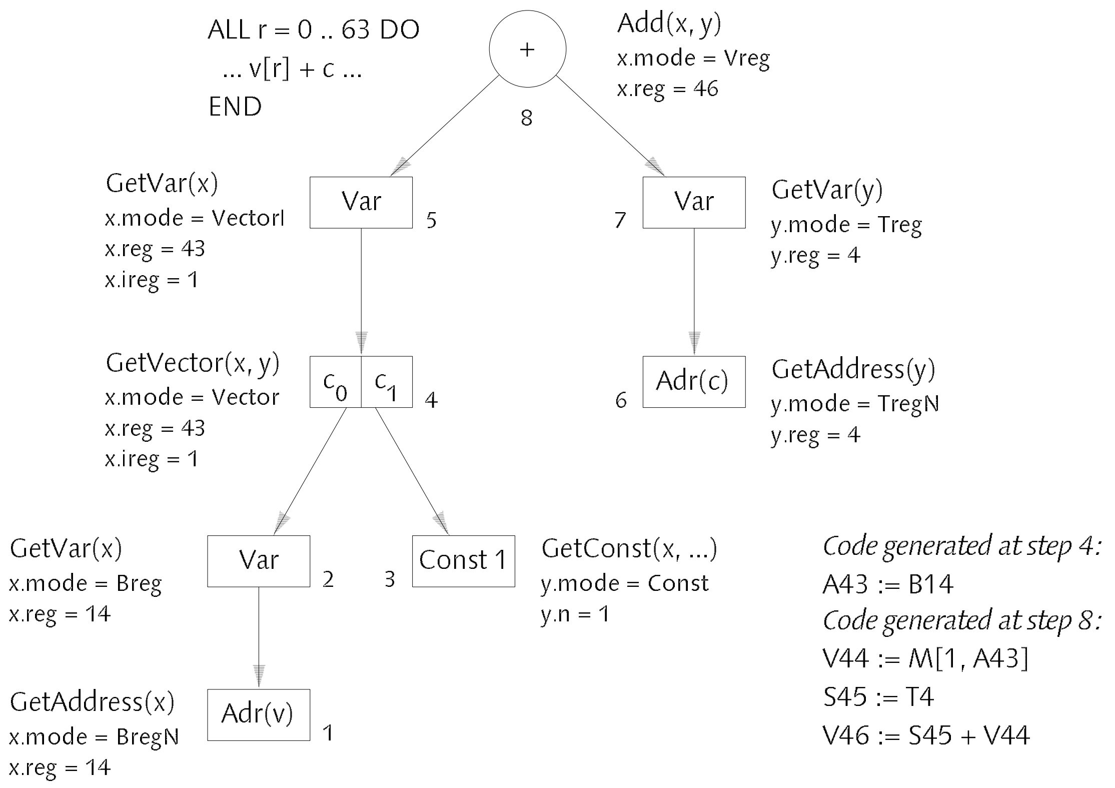


Figure 5.15 Transition of Item Modes for Scalar Addition

For each operation of the language Oberon-V, a procedure is provided by module OVE. Each of these procedures usually accepts one or two item arguments *x* and *y* describing the (location of the) operands. After execution of a particular procedure, the item *x* describes the computed value (the value of y is undefined).

```
PROCEDURE GetAddress (VAR x: Item; adr: ↑Address);
PROCEDURE GetConst (VAR x: Item; con: ↑OVT.Const);
PROCEDURE GetVar (VAR x: Item);
PROCEDURE GetVector (VAR x, stride: Item);
...
PROCEDURE Abs (VAR x: Item);
PROCEDURE Ord (VAR x: Item);
...
PROCEDURE Add (VAR x, y: Item);
PROCEDURE Sub (VAR x, y: Item);
...
```

Procedure *GetVector* constructs a *vector item* with *mode* = *Vector* from its address (*x*, *stride*). The items for *x* and *stride* are directly obtained by evaluating the c_0 and c_1 coefficients of the corresponding range object (cf. previous section, procedure *CExpression*). Note that at this stage of code generation, only range objects with two coefficients occur. If an OVE procedure is called with vector items as arguments, the corresponding vector instructions are generated instead of scalar code. Fig. 5.16 shows the OVE procedure calls, the transition of item modes and the generated code for the addition of a vector and a scalar value.



Code generated at step 4:
A43 := B14
Code generated at step 8:
V44 := M[1, A43]
S45 := T4
V46 := S45 + V44

Figure 5.16 Transition of Item Modes for Vector Addition

Note the special interpretation of the constant stride in step 4: Since vectors with stride 1 can be loaded with special *vector load* instructions which require only a single register argument (the start address), no A-register for the stride is introduced. Instead, it is represented by the special register number 1 which denotes the immediate value 1.

5.3.3 OVV: The Virtual Machine Code

The module OVV contains data structures and procedures to store and optimize virtual machine code. In the sequel two optimizations applied to this virtual code are described briefly. For clarity the instructions of the hypothetical machine introduced in Section 1.1 are used. The basic difference when compared with the virtual instructions used for the Cray Y-MP is the restriction to 2 register files (the S- and V-registers) instead of 5 register files. Since the optimizations described here can be applied both to scalar and vector instructions, it is concentrated on scalar instructions only.

Common Subexpression Elimination, or CSE for short, is the elimination of redundant computations, i.e. the elimination of instructions which calculate an expression that has been calculated before. CSE can be done on different levels during the compilation process but is most effective when applied to a program representation which closely mirrors the available machine instructions. This is due to the fact that translation itself may introduce new common subexpressions which were not apparent when CSE would be applied earlier (e.g. address computations for array elements). The CSE algorithm used in module OVV is based on a well-known technique called *value numbering* [Cocke 1970]. As an example, let us consider the code generation for the expression A[i+1, j] + A[i+1, k] where A is a 100 x 100 real array and i, j and k denote integer variables. The algorithm works as follows: For every virtual instruction computing a new *value*, a new destination register is allocated (which also explains the name *value numbering*). This preserves all previously computed values and hence they can be accessed in any subsequent computation. For the first term A[i+1, j] the following code is generated:

```
;       S0                  address of A
;       S1                  variable i
;       S2                  variable j
;       S3                  variable k
;
        S10 := 1 + S1       ; i+1
        S11 := 100 * S10    ; (i+1)*100
        S12 := S2 + S11     ; (i+1)*100 + j
        S12 := M[S0 + S12]  ; A[i+1, j]
```

Before generating a new instruction, the already generated instructions are searched for an equivalent instruction. If such an instruction is found, no code is emitted but the destination register of the already existing instruction is used instead. The following example illustrates the code generation for A[i+1, k]:

```
;       S13 := 1 + S1       1+S1 has been computed already: use S10
;       S13 := 100 * S10    100 * S10 has been computed already: use S11
;
        S13 := S3 + S11     ; (i+1)*100 + k
        S14 := M[S0 + S13]  ; A[i+1, k]
        S15 := S12 +F S14   ; A[i+1, j] + A[i+1, k]
```

In the example the expression i+1 must be computed to determine the array offset for the second term. Scanning backwards in the instruction stream reveals that the same computation has been performed already, and that the result is in register S10 (and must still be there because of the consecutive register numbering). Instead of generating an add instruction and introducing a new register, the value in register S10 is used. The same happens, when an attempt is made to compute (i+1) * 100. The value i+1 is in S10, thus a multiply instruction 100*S10 should be generated. Scanning backwards reveals that the result of 100 * S10 is held already in S11; thus no code must be emitted but S11 can be used instead. When continuing this way, all redundant instructions will be eliminated and optimal code (as far as CSE is concerned) results.

Further improvement is achieved, by a special treatment of commutative operations: Obviously, an instruction S10 * S11 yields the same result as S11 * S10. It is a good idea to *normalize* instructions for commutative operations, e.g. such that the first register operand always has the smaller register number. Often, such a normalization is required anyway in order to have an immediate operand at the right position in the instruction format.

In OVV, the virtual code is simply stored in an array. Each array element corresponds to a virtual instruction. The element index can be regarded as *program counter* and also corresponds to the destination register number, which therefore has not to be stored explicitly. Note that the CSE algorithm as it has been presented here must be applied within *extended basic blocks* only, i.e. no label must occur within the code stretch to be optimized, except at its very beginning. Furthermore, special precautions are required for load and store instructions; these topics are not discussed here. CSE would be relatively slow without optimizations: Before the i-th instruction can be generated, all i–1 previous instructions in the basic block must be inspected and compared with the new instruction in question. For a sequence of n redundancy free instructions, $(n^2-n)/2$ comparisons are necessary in total; i.e. the algorithm has a run-time complexity of $O(n^2)$. In OVV, all instructions denoting the same operation are chained in the instruction array. Thus, searching backwards requires only searching instructions denoting the same operation. In practice an almost linear run-time is achieved.

To give an idea of the implementation, the procedure *Op2* and parts of the required data structures are shown below. Op2 is responsible for binary operations *op* applied on two operands held in virtual registers *j* and *k*. The result of the computation is assigned to the virtual register *i* (corresponding to the instruction index in the *Code* array). Op2 is used in module OVE whenever a binary operation has to be generated. The *Root* array contains the roots of all instruction chains linking instructions with the same *op* field.

```
CONST
  NofInstr = ... ; CodeSize = ... ;
```

```
VAR
   Root: ARRAY NofInstr OF INTEGER;
   Code: ARRAY CodeSize OF RECORD
      op, j, k, ... : INTEGER; (* instruction Sj op Sk *)
      link: INTEGER; (* previous instruction with same op field; –1 denotes end of chain *)
      ...
   END;

PROCEDURE Commutative (op: INTEGER): BOOLEAN; (* TRUE if op is commutative *)
PROCEDURE Insert (op: INTEGER; VAR i: INTEGER; j, k: INTEGER); (* inserts new instr. *)

PROCEDURE Op2 (op: INTEGER; VAR i: INTEGER; j, k: INTEGER);
BEGIN
   IF (j > k) & Commutative(op) THEN i := j; j := k; k := i END; (* canonical form: j <= k *)
   i := Root[op];
   WHILE (i >= 0) & ((Code[i].j # j) OR (Code[i].k # k)) DO i := Code[i].link END;
   IF i < 0 THEN (* no CSE found *) Insert(op, i, j, k) END
END Op2;
```

As a final example for CSE its application to *extended basic blocks* is shown: Since jumps do not invalidate the current contents of the virtual registers, their contents may also be reused after a jump, i.e. *across* basic block boundaries. Thus, a statement sequence in a THEN branch may also profit from common subexpressions in the conditional expression of an IF statement. In the example below, the code in the THEN branch is reduced to a single instruction only. Also redundant *index checks* are eliminated this way in OVV (not shown here).

```
            IF A[i, j] < x THEN A[i, j] := x END

;           S0                    address of A
;           S1                    variable i
;           S2                    variable j
;           S3                    variable x
;
            S10 := 100 * S1       ; i*100
            S11 := S2 + S11       ; i*100 + j
            S12 := M[S0 + S11]    ; A[i, j]
            S13 := S12 <F S3      ; A[i, j] < x
            Jump End (~S13)       ; IF A[i, j] < x THEN
            M[S0 + S11] := S13    ; A[i, j] := x
End         ...                   ; END
```

The second optimization performed in OVV is *Instruction Scheduling* or *IS* for short: In *pipelined* computers the execution of an instruction is divided into several suboperations that are executed sequentially on several independent units, called *pipestages*. With this organization, multiple instructions may execute concurrently and each instruction occupies a different pipestage (Figure 5.17). At time t instruction a is in pipestage 1, at time t+1 instruction a is in pipestage 2 and b is in pipestage 1, etc.

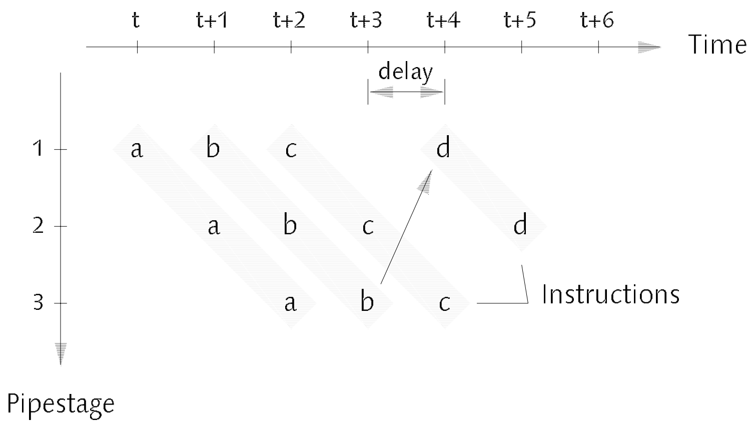


Figure 5.17 Pipelined Execution of Instructions

An instruction depending on the result of a previous instruction must not be issued until the required result is available. That is, either the compiler ensures that the instruction is not issued too early or the processor delays the instruction until its operands become available (*pipeline interlock*). In Figure 5.17, instruction d is assumed to use an operand (a register) computed by instruction b and thus cannot be issued at time t+3 but must wait until instruction b has terminated at t+4. Pipeline interlocks due to such *timing hazards* may significantly decrease the performance of a program. On Cray machines (cf. [Robbins 1987]) and many popular RISC architectures (e.g. MIPS [Kane 1987] or SPARC [Sun 1987]) instructions are executed in a such a pipelined fashion and hence a compiler should carefully avoid pipeline interlocks imposed by the generated code. This is achieved by *instruction scheduling*, i.e. the reordering of instructions such that interlocks are avoided but the effect of the program is preserved. In the example it were possible to swap the instructions a and b, assuming that a and b are independent. This would reduce the execution time of the four instructions a ... d by 1 clock cycle (Figure 5.18).

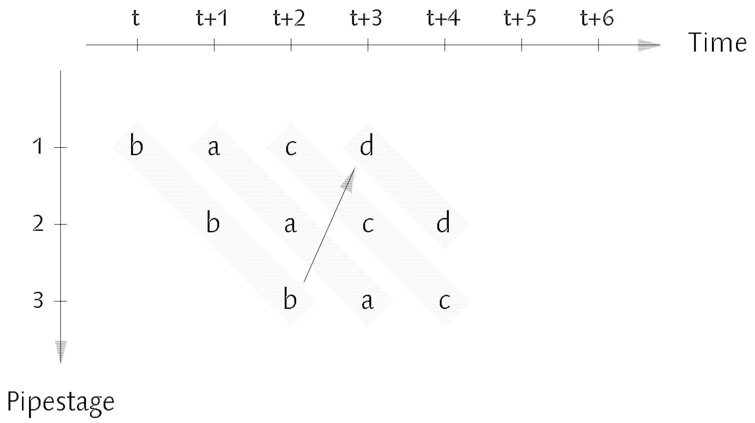


Figure 5.18 Pipelined Execution of Instructions after Scheduling

The IS problem is well-studied in the literature and approaches for its solution go back to methods originally developed for microcode compaction [Landskov 1980]. A thorough survey of research can be found in [Gross 1983] or [Hennessy 1983]. Like many practical problems, IS is NP-complete (see e.g. [Gross 1983]) and hence only heuristic solutions are likely to be practicable. Register allocation may introduce additional dependences that can restrict the possibilities for instruction reordering. On the other hand, IS may increase *register pressure*, i.e. the number of registers that are alive at a specific time, and hence complicates register allocation. Thus, register allocation and IS pursue conflicting goals. There are pre- and postpass solutions: the former reorder instructions before register allocation is done (e.g. [Auslander 1982], one of the approaches described in [Gross 1993] or the solution used here), the latter perform IS after register allocation (e.g. [Gibbons 1986], [Davidson 1984] and [Davidson 1986]). Recent work tries to improve the situation by using information of the scheduling phase to drive register allocation [Bradlee 1991].

The most common IS algorithms are variations of *list scheduling* ([Landskov 1980], [Gibbons 1986], [Warren 1990]): A *dependence graph* is constructed for each basic block. Nodes of the graph represent machine instructions and edges denote dependences; an edge from node *a* to node *b* indicates that *a* must be executed before *b* in order to preserve the *overall effect* of the basic block. The set of all nodes that have no predecessor, i.e. the set of instructions that depend on no other instruction, is called the *candidate set* (Figure 5.19).

```
S10 := S0 + S1
S11 := 2 * S10
S12 := S4 – S3
S13 := 10 + S12
S14 := S11 * S13
S15 := S10 / S12
S16 := S5 * S11
S17 := S14 / S15
```

(S0 ... S5 are assumed to be available already)

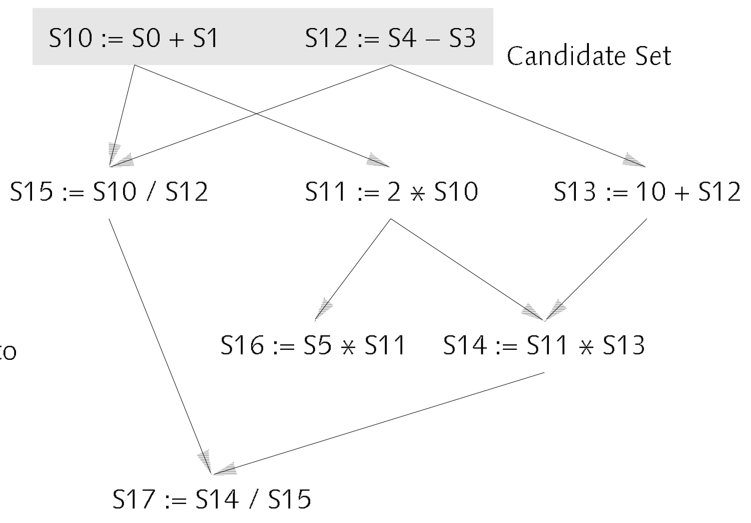


Figure 5.19 Dependence Graph and Candidate Set for a Basic Block

The list scheduling algorithm corresponds to a topological sort of the graph nodes and proceeds as follows: While the candidate set and hence the graph is not empty, a suitable instruction *i* of the candidate set is selected according to some heuristics and its node is removed from the graph. Note that any instruction *j* that had instruction *i* as its single predecessor, has no predecessor after removing *i* and then enters the candidate set. A simple framework for list scheduling can be formulated in Oberon-V:

```
PROCEDURE Schedule (G: Graph; Select: PROCEDURE (G: Graph): Node);
   VAR n: Node;
BEGIN
   WHILE ~Empty(G) DO
      n := Select(Candidates(G));
      Remove(G, n);
      Emit(n)
   END
END Schedule;
```

The quality of a list scheduling algorithm depends on the selection heuristic. Although list scheduling has a worst-case run-time proportional to O(n^2) where n is the number of instructions, a linear behavior can be expected for practical cases [Gibbons 1986].

List scheduling appears to be a relatively simple technique for instruction scheduling: basically, the dependence graph corresponds to the expression tree drawn upside down (see Figure 5.19). Nevertheless, a simpler method was developed for the Oberon-V compiler [Griesemer 1992]. The method profits from the representation of virtual code by an array: For IS, instead of appending consecutive instructions, they are *inserted* in the code array at the earliest possible location determined by data dependences and possible interlocks. Using the array representation mentioned for CSE, the program counter can be taken as the destination register number. Assuming that a new instruction is issued every clock cycle (without interlocks), the program counter may also be interpreted as the *current clock*. This leads to a surprisingly simple method for directly placing an instruction while observing its dependences.

Let I(a, b) be an instruction depending on the contents of registers a and b (i.e. the results of the instructions a and b). When E(a) and E(b) are the execution times of these instructions, a+E(a) and b+E(b) are the times when each result is available. Consequently, both results are available at Max(a+E(a), b+E(b)), which is the earliest possible location for I.

Let us assume an execution time of 2 cycles for addition and memory load and 3 cycles for multiplication (there are much longer execution times for Cray Y-MP instructions). In the array expression example, the computation 100 * S10 depends on S10 and the earliest issue time is 12 = Max(0, 10+2), since the time used to compute S10 is 2 cycles and the operand 100 is available immediately. Instead of simply appending the instruction as before at position 11, it is inserted at position 12. Note that the destination register number is automatically determined by the instruction position and has not to be stored. The resulting *free instruction slots* at position 11 may be thought as being filled with a *no-op* instruction. Continuing this way, the following instruction sequence for A[i+1, j] is obtained:

```
;       S0                  address of A
;       S1                  variable i
;       S2                  variable j
;       S3                  variable k
PC                                               Execution time   Result available
10      S10 := 1 + S1       ; i+1                2                12
11      no-op
12      S12 := 100 * S10    ; (i+1)*100          3                15
13      no-op
14      no-op
15      S15 := S2 + S12     ; (i+1)*100 + j      2                17
16      no-op
17      S17 := M[S0 + S15]  ; A[i+1, j]          2                19
```

Hence, whenever a new instruction has to be inserted, its earliest issue time is determined due to its data dependences and the instruction is then inserted at the calculated position. Inserting an instruction after the end of the current code block extends the block. When inserting an instruction in the middle of the code block, the instruction replaces the *next* free instruction slot (possibly at the end of the code block). Eventually, the following code for A[i+1, j] + A[i+1, k] is obtained:

```
;       S0                  address of A
;       S1                  variable i
;       S2                  variable j
;       S3                  variable k
PC                                              Execution time  Result available
10      S10 := 1 + S1       ; i+1               2               12
11      no-op
12      S12 := 100 * S10    ; (i+1)*100         3               15
13      no-op
14      no-op
15      S15 := S2 + S12     ; (i+1)*100 + j     2               17
16      S16 := S3 + S12     ; (i+1)*100 + k     2               18
17      S17 := M[S0 + S15]  ; A[i+1, j]         2               19
18      S18 := M[S0 + S16]  ; A[i+1, k]         2               20
19      no-op
20      S20 := S17 +F S18   ; A[i+1, j] + A[i+1, k]
```

The rules are a little bit more complex if store instructions are involved: To preserve the effect of a code sequence, the relative ordering of consecutive loads and stores must remain unchanged. These problems are not discussed here.

After generating the virtual code for an extended basic block, it must be translated into the target machine code. That is, for each virtual instruction one (or several) target machine instructions have to be selected and final register allocation must be done. When register pressure is too high, *spill code* has to be inserted. In OVV this translation is done by an additional sweep over the virtual code. A brief overview on this translation process is given in the sequel:

Target instruction selection: Since the virtual instructions differ from the target machine instructions only in the unbounded number of registers used, this part of the translation process is trivial: When the final register numbers are known, the final machine instructions can be generated directly.

Register allocation: At this level only registers for temporaries occurring during expression evaluation have to be allocated. In the OV compiler, the A-, S- and V-registers are used for this purpose. During translation of the virtual code into final machine code, whenever a new result is computed, an appropriate physical register is allocated from these register files using a simple round-robin algorithm. The physical register number is remembered together with the corresponding virtual register. Whenever the virtual register is referenced again, the physical register number is used instead. At some point, the virtual register is not referenced any more and the corresponding physical register should be released. To detect this situation, a simple technique which has been described in [Freiburghouse 1974] is used. The key idea is to assign a reference counter to every virtual register, i.e. instruction in the array representation. During generation of the virtual code, this reference counter is increased whenever the corresponding register is referenced by another instruction. During target code generation, whenever a virtual register is referenced, the corresponding reference counter is decreased. When it drops to zero, this was the last reference to the virtual register and hence the corresponding physical register can be released.

Spill code generation: When straight-forward code is generated for expressions (without CSE or IS), very few registers are usually required to hold temporaries. However, CSE and IS significantly raise register pressure. Even when reserving eight or more registers for expression evaluation, excessive register demand may require to spill currently unused registers to memory. When spilling to memory, reload instructions have to be inserted which may impose new interlocks.

In case of the Cray Y-MP, spilling is relatively cheap because a good portion of the temporary B- and T-register files is reserved for this purpose. Load and store accesses to temporary registers are executed in one cycle, therefore no additional interlocks are introduced. Spilling to main memory would be impractical due to the long latency of load instructions. Note that only *local spilling* is necessary, i.e. within basic blocks. This is in contrast to simple Chaitin-like graph-coloring techniques for register allocation which globally (i.e. for an entire procedure) decide, which values have to reside in memory [Chaitin 1982]. Nevertheless, the decision which register to spill is difficult and depends highly on the context in which the register is used. As a simple heuristic, in OVV the physical register which has not been used for the longest time is spilled.

5.4 Hardware Support

Some problems related to code generation for the particular instruction set of the Cray Y-MP have not been discussed yet. In fact, this instruction set has some deficiencies which make the generation of correct code cumbersome. It is interesting that most of these deficiencies apply also to an older architecture, namely the CDC 6000 series (already pointed out in [Wirth 1972]). This relationship may not surprise since Seymour Cray worked also on the design of the CDC 6600 and CDC 6700 machines. For the sake of brevity the details are not discussed but the main trouble spots are summarized only:

1. *No overflow check on integer arithmetic*. For both, the 32 bit and 64 bit registers, there is no hardware support to detect integer overflows. The price for guarding all integer operations is prohibitive.

2. *No compare instructions*. Conditional jumps depend on the comparison of the A0- or S0-registers with zero. A general comparison of the form x *rel* y is transformed to (x–y) *rel* 0. If overflow occurs in the subtraction (which is not simply detectable), the result of the relational expression is wrong.

4. *No arithmetic shift instructions*. Since also no instruction for sign extension is available, an arithmetic shift right instruction must be emulated by a lengthy sequence of logical and arithmetic operations.

5. *No support for 64-bit integer multiplication/division*. 64-bit integer multiplication and division is performed using floating point arithmetics. Since full 64-bit results would require subroutine calls or inlined code sequences of up to 30 instructions for multiplication and even more instructions for division, currently only 46-bit multiplication and division are implemented in Oberon-V.

Besides these deficiencies, the Cray Y-MP is quite clean and orthogonal, thus making a clean and small code generator possible. Nevertheless, from the viewpoint of the compiler writer, we would like to propose a single major point that should be improved (increased), namely the *number of general purpose registers*. Each of the three general-purpose register files (A-, S- and V-registers) provides only 8 registers. Furthermore, the A0 and S0 registers can be read by a few instructions only, thus making them worthless for expression evaluation. When common subexpression elimination and instruction scheduling is performed, register pressure is quite high and frequently spilling is required. Hence, it is difficult to allocate local variables in these register files. Currently, such variables are held in the two temporary register files (B- and T-registers), thus requiring additional register-register moves if they are accessed. Larger general-purpose registers (and the omission of the temporary registers) would simplify the code generator and probably also improve the execution speed of the generated code.

6 Measurements

Some empirical results on a small collection of typical subroutines and programs are presented. All the measurements have been done on a Cray Y-MP [Cray 1988]. If not specified otherwise, the results are given in clock cycles, the cycle time is 6ns. Although only average values are presented here, the results may not be very accurate: since the Cray Y-MP is a multi-processor shared-memory system, a running process may be delayed because of memory bank conflicts caused by other processors. The estimated relative error is 5–10%.

6.1 BLA Subprograms

The *Basic Linear Algebra Subprograms* or *BLAS* for short [Lawson 1979] have been very successful and are used in a wide range of numerical software including packages such as Linpack (see Section 6.2). They have become a de facto standard for elementary vector operations. In the following a collection of measurements made for four frequently used BLA procedures, namely *sscal*, *saxpy*, *sasum* and *sdot* is shown (cf. Appendix B.1). In Oberon-V these procedures reduce to simple "one-liners" which preferably are used directly instead of calling the subroutine.

Table 6.1 shows the measurements for these four BLA procedures, different vector lengths (*n*) and different (combinations of) compiler options: */x* means without index checks, */i* turns instruction scheduling (IS) off, */c* disables common subexpression elimination (CSE) and */ic* is without CSE and IS. All times are given in clock cycles per element. The right-most column shows the results obtained with the Cray Fortran compiler cft77. To obtain comparable results, a one-to-one translation of the Oberon-V BLA procedures into Fortran subroutines but not the Fortran BLAS library has been used, because this library is partially hand-tuned. The Fortran compiler also generates vector instructions for these subroutines. Furthermore, different optimizations including CSE and IS are performed, but no index checks are generated by cft77.

a)

n	*sscal*	*sscal/x*	*sscal/i*	*sscal/c*	*sscal/ic*	*Fortran*
10	31.7	20.2	38.3	48.0	28.5	23.1
100	4.1	2.5	4.0	5.7	3.5	3.1
1'000	1.9	1.5	1.7	1.7	1.6	1.6
10'000	1.6	1.5	1.5	1.5	1.4	1.5
100'000	1.6	1.5	1.7	1.5	1.4	1.5

b)

n	*saxpy*	*saxpy/x*	*saxpy/i*	*saxpy/c*	*saxpy/ic*	*Fortran*
10	42.3	26.2	46.8	60.2	44.3	17.9
100	5.0	3.6	5.1	5.6	5.3	3.4
1'000	1.8	1.8	1.9	2.0	2.2	1.6
10'000	1.6	1.5	1.6	1.5	1.6	1.5
100'000	1.5	1.5	1.8	1.5	1.5	1.5

c)

n	*sasum*	*sasum/x*	*sasum/i*	*sasum/c*	*sasum/ic*	*Fortran*
10	183.9	165.3	189.5	181.4	176.1	43.3
100	20.7	18.5	23.0	20.6	19.5	6.3
1'000	3.9	3.7	3.9	4.6	3.8	1.5
10'000	2.4	2.3	2.3	2.6	2.3	1.2
100'000	2.1	2.1	2.1	2.3	2.1	1.1

d)

n	*sdot*	*sdot/x*	*sdot/i*	*sdot/c*	*sdot/ic*	*Fortran*
10	185.1	174.0	189.7	189.9	187.6	51.2
100	21.1	19.8	20.9	21.2	21.0	7.5
1'000	4.3	4.0	4.3	4.2	4.2	1.7
10'000	2.4	2.5	2.6	2.5	2.5	1.3
100'000	2.3	2.3	2.5	2.3	2.4	1.2

Table 6.1 Execution Times for BLA Subroutines (vector code, in clock cycles/element)

The measurements include the times for the procedure calls and, notably, also for index checks of arguments (constructed arrays, see Section 3.9). Thus, the execution times for vectors with small n ($n \leq 100$) are completely dominated by the procedure call overhead (see also Table 6.2). The best results are obtained when index-checks are disabled. However, because index checks are performed only once before executing the ALL statement, for larger n no significant difference is noticed anymore. Also the effect of the optimizations disappears for large n, which is a clear indication that they influence mostly the (scalar) code for procedure calls, index checks and loop initialization. When compared with Fortran, for n ≥ 1000 the same or even better results are obtained for sscal and saxpy (Table 6.1 a) and b). In case of sasum and sdot (Table 6.1 c) and d) however, only half the execution speed of the corresponding Fortran code is achieved. It seems that this difference is mainly due to a better but also more complicated code pattern for reduction loops generated by the Fortran compiler (the Fortran code proceeds in steps of 128 instead of 64 vector elements and uses *two* vector registers for the partial sums; see also Section 3.8.2).

Table 6.2 shows the timings for the same BLA procedures, when no vector instructions are used (compiler option */v*) but scalar code is generated for the ALL statements and the SUM function. The numbers in the right-most column (*loops*) have been measured for conventionally implemented BLA procedures using WHILE loops instead of ALL statements.

n	*sscal/v*	*sscal/vx*	*sscal/vi*	*sscal/vc*	*sscal/vic*	*loops*
10	68.3	66.1	84.1	67.0	88.9	104.1
100	43.0	43.1	58.9	45.3	63.1	77.4
1'000	40.5	40.7	56.6	42.8	60.4	74.6
10'000	40.3	40.4	56.2	42.4	60.2	74.3
100'000	40.5	40.4	56.2	42.4	60.1	74.3

a)

n	*saxpy/v*	*saxpy/vx*	*saxpy/vi*	*saxpy/vc*	*saxpy/vic*	*loops*
10	83.9	116.3	115.2	87.4	123.8	130.4
100	52.6	56.1	83.6	55.5	89.3	102.6
1'000	49.5	50.5	80.6	52.4	85.8	99.6
10'000	49.2	50.0	80.4	52.0	85.4	99.4
100'000	49.1	49.6	80.4	52.0	85.3	99.3

b)

n	*sasum/v*	*sasum/vx*	*sasum/vi*	*sasum/vc*	*sasum/vic*	*loops*
10	59.6	57.8	67.3	75.4	76.1	116.1
100	43.3	43.0	51.3	43.2	53.9	77.2
1'000	41.7	41.5	49.6	41.3	51.5	73.9
10'000	41.5	41.4	49.2	41.3	51.3	73.7
100'000	41.4	41.3	49.4	41.9	51.3	73.6

c)

	n	*sdot/v*	*sdot/vx*	*sdot/vi*	*sdot/vc*	*sdot/vic*	*loops*
	10	84.6	83.9	94.1	91.3	99.0	123.5
	100	60.5	63.4	69.7	61.1	71.6	103.5
	1'000	57.9	62.1	67.1	58.1	68.8	101.5
	10'000	57.6	60.6	66.7	57.9	68.5	101.3
d)	100'000	57.4	60.5	66.5	57.5	68.4	101.3

Table 6.2 Execution Times for BLA Subroutines (scalar code, in clock cycles/element)

First of all, when comparing these numbers with Table 6.1, the tremendous speed-up obtained by using vector instructions becomes obvious. In this case the BLA procedures are about 25 times faster than when using scalar instructions only. Secondly, even if no vector instructions are available, the scalar code for ALL statements is about 1.85 times faster then the code generated for conventional WHILE loops (the *x/v* column has been compared with the *loops* column for n = 100'000). This improvement is due to the fact that usually no index checks are performed within the ALL statement but only a few checks are necessary before executing the ALL statement (Section 3.9). Furthermore, the translation scheme for ALL statements automatically reduces expensive address computations for array elements to pointer arithmetics (Section 3.7).

For scalar code, also the effect of common subexpression elimination and instruction scheduling becomes significant: CSE alone yields a speedup of 1.34 (columns /v and /vi for n = 100'000). The effect of IS alone is marginal (1.03, columns /v and /vc for n = 100'000) but together a speedup of 1.41 is achieved (columns /v and /vic for n = 100'000). The small effect of IS is due to the relative short basic blocks in all four procedures.

6.2 Linpack Subroutines

Linpack is a collection of Fortran subroutines which analyze and solve various systems of simultaneous linear algebraic equations [Dongarra 1979]. Two Linpack procedures have been translated one-to-one from Fortran to Oberon-V and their performance has been measured. The procedures *sgefa* (*s*ingle precision *ge*neral *fa*ctorization) and *sgesl* (*s*ingle precision *ge*neral *sol*ver) are used to solve n simultaneous linear equations Ax = b where A is an n by n matrix and x and b are two vectors of length n. The corresponding Oberon-V procedures may be found in Appendix B. Tables 6.3 and 6.4 show *normalized* execution times in clock cycles, i.e. the execution times have been divided by the following values:

for sgefa: $n^3/3$
for sgesl: n^2

These values roughly estimate the number of "axpy-operations" executed by sgefa and sgesl, i.e. the number of assignment statements of the form y[i] := a∗x[i] + y[i]. For example, a value 1.8 indicates that 1.8 clock cycles have been used on average for an axpy-operation.

n	*sgefa*	*sgefa/x*	*sgefa/ic*	*Fortran*
100	6.3	5.0	11.8	3.7
200	3.8	3.2	6.5	2.7
300	3.1	2.6	4.9	2.5
400	2.7	2.3	4.0	2.4
500	2.4	2.1	3.5	2.3
600	2.3	2.0	3.2	2.3
700	2.2	2.0	2.9	2.3
800	2.1	1.9	2.8	2.2
900	2.0	1.8	2.7	2.2
1000	2.0	1.8	2.5	2.2

Table 6.3 Execution Times of sgefa (in clock cycles/axpy-op)

If index checks are omitted (*sgefa/x*) an average speedup of 1.15 is achieved. Again, for larger n the effect of index checks becomes smaller. On the other hand, the effect of CSE and IS is quite high: both optimizations yield an average speedup of almost 1.47 (for all n, columns *sgefa* and *sgefa/ic*). When compared with Fortran, the Oberon-V code (*sgefa/x*) is slower for small n ($n < 300$) but the same or even better performance is achieved for larger n. In general, much better scalar code is generated by the Fortran compiler. This also explains the leading position of Fortran for small n; an observation made before (Section 6.1). However, a detailed investigation of these effects is beyond the scope of this thesis.

n	*sgesl*	*sgesl/x*	*sgesl/ic*	*Fortran*
100	6.2	4.8	10.2	3.5
200	3.8	3.1	5.8	2.3
300	3.1	2.5	4.3	1.9
400	2.7	2.3	3.6	1.7
500	2.4	2.1	3.2	1.6
600	2.2	2.0	2.9	1.5
700	2.2	1.9	2.7	1.5
800	2.1	1.9	2.6	1.4
900	2.0	1.8	2.5	1.4
1000	1.9	1.8	2.4	1.4

Table 6.4 Execution Times of sgesl (in clock cycles/axpy-op)

In Table 6.4 essentially the same results as in Table 6.3 can be observed. The improvement obtained by omitting index checks decreases for large n and the speedup gained by CSE and IS is 1.35 on average. For n = 1000, every 2 clock cycles an axpy operation containing two floating-point operations is performed by sgefa. For a cycle time of 6ns this corresponds to a performance of 167MFlops. Together with sgefa, a system of 1000 simultaneous linear equations can be solved in approximately 4s.

In Fortran arrays are mapped to memory in *column-major order*, i.e. consecutive elements of a matrix column are mapped to consecutive memory words. Thus, if an algorithm processes a matrix column-wise, usually the vector stride is one. In Oberon-V arrays are mapped in *row-major order* and hence the stride of a column vector is usually greater than one. Accessing such vectors may lead to *memory bank conflicts* and hence to significant *delays* (for an introduction into interleaved memory systems see for example [Hennessy 1990]). Indeed, the Linpack subroutines essentially *rely on the array mapping of Fortran* and they perform badly for certain values of n if they are translated to Oberon-V in a one-to-one way. However, instead of rewriting all algorithms such that they process rows instead of columns, the same effect can be obtained easily in Oberon-V by *transposing* the matrix using an array constructor:

```
PROCEDURE Work (VAR a: ARRAY OF ARRAY OF REAL, ...);
(* do anything with a *)
END Work;

BEGIN ...
  Work([i, j = 0 .. LEN(a)−1: a[j, i]], ...); ...
```

If a certain array mapping is required within a procedure (e.g. column-major order instead of row-major-order), the appropriate array is simply passed as a parameter. When the procedure is called, the desired mapping (e.g. a transposition) is performed using an array constructor. Note that this does not mean that the array is copied, but only that the array mapping is changed which does not impose any additional run-time costs. Using this technique, various kinds of memory mappings can be tested without changing an algorithm.

6.3 Compiler Data

Finally, a survey of the size and the compilation speed of the Oberon-V compiler OV is given. Table 6.5 illustrates the sizes of individual modules in (non-empty) lines of code (*Lines*) and in the number of statements (*Stats*). The size of the object code for the Ceres-2 workstation [Heeb 1988] containing a NS32532 CPU is given in KB. The right-most column shows the object code sizes of corresponding modules of the Oberon-2 compiler NOP2 for the same machine [Mössenböck 1991][Crelier 1990]. Apparently, when ignoring module OVV, both compilers have the same size. In contrast, the size of the Cray Fortran compiler cft77 is 4.7MB which is more than 50 times the size of the Oberon-V compiler.

Module	*Functionality*	*Lines*	*Stats*	*Code*	*NOP2*	
OVS	Scanner	300	330	3.3	3.4	(NOPS)
OVT	Table Handler	1670	1470	23.6	24.9	(NOPT + NOPB)
OVP	Parser	770	850	9.3	11.4	(NOPP)
OVO	Object Files	430	470	7.7	8.8	(NOPL)
OVV	Virtual Code	700	790	16.2	–	–
OVE	Expressions	1400	1400	18.7	17.4	(NOPC)
OVC	Tree Traversal	620	620	10.1	6.6	(NOPV)
OV	User Interface	80	80	1.3	2.0	(NOP2)
	Total	5970	6010	90.2	74.5	

Table 6.5 Size of the Oberon-V compiler OV (in KB)

In order to determine the compilation speed of the Oberon-V compiler, procedures of a longer module (containing a collection of Linpack procedures) have been removed step-by-step and the time for its compilation with respect to different compiler options has been measured (Table 6.6). Overall, the compilation time is nearly linear in the program length for all options (the linear regression coefficient r is 0.98). Surprisingly, if CSE is disabled (option */c*) the compilation speed decreases! This is due to the fact that the compilation time is also linear in the length of the generated code. For modules with many common subexpressions as it is the case here, CSE significantly reduces the length of the generated code and thus also the compilation time. In the normal case (*no options*), a compilation speed of about 120 lines/s is achieved on the Ceres-2 (NS 32532, 25MHz) and 620 lines/s on a DECstation 5000 (MIPS R3000, 25MHz). The Cray Fortran compiler translates up to 300 lines/s on the Cray Y-MP (167MHz).

Lines	*no options*	*/v*	*/i*	*/c*	*/ic*
1146	10.1	10.1	10.1	11.7	11.7
1143	8.3	8.3	8.3	9.4	9.4
828	6.0	6.0	6.1	6.9	6.9
501	3.8	3.8	3.9	4.3	4.4
321	2.6	2.6	2.6	3.0	3.0
186	1.7	1.7	1.7	1.9	1.9

Table 6.6 Compilation Speed for different Compiler Options (in s)

7 Summary and Conclusions

Section 7.1 gives a brief survey of our achievements, whereas Section 7.2 points out some features that could be improved. Finally, conclusions are draw in Section 7.3

7.1 What has been achieved

We have introduced two language constructs, namely the *ALL statement* and a new form of *array constructors*, both oriented towards the programming of numerical applications for vector computers. The constructs have been integrated into a new programming language called Oberon-V, a descendant of the programming language Oberon [Wirth 1988a]. A possible implementation scheme has been described and an experimental Oberon-V compiler for the Cray Y-MP vector computer has been constructed. We have implemented a few typical applications in Oberon-V and have compared their performance with corresponding Fortran solutions.

The ALL statement allows to express directly *independence* of a sequence of assignment statements and thus can always be translated into vector instructions, as long as no conditional expressions occur. Unfortunately, a compiler cannot check in general whether assignments enclosed in an ALL statement are independent or not. Thus, it is the programmer's responsibility to prove its correct usage. In our experience, for practical cases the required independence property is rather easy to show or even obvious. For difficult cases however a compiler will not be able to solve the problem instead. We may compare this situation with the use of a conventional loop where (in the general case) also the programmer alone can ensure its termination.

The provability of programs cannot be measured objectively. However, the correctness of programs using ALL statements is easier to prove than the correctness of programs using corresponding loops, because overspecifications such as the execution order or additional statements for incrementing induction variables can be avoided. Furthermore, since a programmer may explicitly state that parts of a program can be executed in parallel, a reader of the program is also directly informed about this fact and thus may gain a deeper insight into an algorithm.

The translation of ALL statements requires neither complicated analysis techniques nor time-consuming optimizations but is obtained by a simple transformation of its internal representation and a second traversal of this data structure for final code generation. For typical applications the execution speed of the generated code is comparable with the execution speed of the code generated by the Cray Fortran compiler. We have implemented additional optimizations on the instruction level, such as common subexpression elimination and a simple instruction scheduling algorithm for scalar instructions. Though the effect of these optimizations tends to be completely overshadowed by the effect of vector instructions as long as only ALL statements are considered. However, we must not conclude that such optimizations are worthless: if only one half of the executed code of a program can be vectorized due to Amdahl's Law, a speedup of a factor two may be achieved in the best case, but only if the vectorized program portion would execute in zero time. A reasonable scalar performance is therefore required to obtain good results.

Array constructors are a powerful and clean construct to specify arbitrary subarrays as procedure arguments. Similar mechanisms exist in other languages, but they usually do not offer the same flexibility or are inacceptably inefficient for the aim pursued here (e.g. the call-by-name mechanism of Algol 60 [Naur 1960]). The implementation of array constructors goes hand-in-hand with the implementation of ALL statements. Indeed, almost the entire compiling machinery introduced for ALL statements has been reused to implement array constructors. Since array constructors can only be used as arguments for open array parameters, they only impose additional costs if the extra flexibility is really needed.

The Oberon-V cross-compiler has been implemented in Oberon and runs under the Oberon System [Wirth 1988b]. It consists of some 6000 lines of code or about 90KB object code for the Ceres-2 workstation [Heeb 1988]. The compilation speed is about 120 lines/s on the Ceres-2 (NS 32532, 25MHz) and 620 lines/s on a DECstation 5000 (MIPS R3000, 25MHz). These figures compare favorably with the Fortran-77 compiler on the Cray with about 4.7MB object code size and a compilation speed of 300 lines/s (Cray Y-MP, 167MHz).

7.2 What could be improved

1. An ALL statement describes a set of assignment sequences that must be independent of each other, i.e. two different assignment sequences must not modify the same variable. Thus, if in a particular sequence a temporary value is computed which is to be used later in the same sequence, an array with the same dimensionality as the ALL statement must be introduced. If it were possible to declare some kind of "local variables" within ALL statements, such an array would be unnecessary. The following example illustrates a possible solution:

```
ALL r = a .. b WITH t: REAL DO
    t := a[r] * b[r];
    b[r] := t * c1;
    c[r] := t + c2
END
```

The identifier t is visible within the scope of the ALL statement only. For different values of the range identifier r, different variables are denoted by t. A similar approach has been chosen in the FORALL statement in Vienna Fortran [Zima et. al. 1992], a Fortran extension oriented towards multiprocessor architectures with distributed memory.

2. We have described the translation of an ALL statement into a sequence of vector or scalar instructions for a single-processor machine. It should not be too complicated to generate code for a multiprocessor machine (e.g. the Cray Y-MP can be equipped with up to eight processors). The key idea is simple: for a machine with n processors, the index set specified by the range declaration of an ALL statement is divided into n subsets, e.g. by dividing the outermost range into n subranges. For example, an ALL statement of the form

```
ALL r = a .. b DO S(r) END
```

had to be divided into n ALL statements

ALL $r_1 = a_1$ .. b_1 DO $S(r_1)$ END
ALL $r_2 = a_2$ .. b_2 DO $S(r_2)$ END
...
ALL $r_n = a_n$ .. b_n DO $S(r_n)$ END

with $a_k = b_{k-1} + 1$ $(2 \le k \le n)$, $a_1 = a$ and $b_n = b$. Each of these ALL statements could be executed on a different processor node. At the end, all processes have to be synchronized and the main processor would take control again. Because no inter-process communication is required during the execution of the ALL statements, a speedup almost proportional to the number of processors could be possible.

3. ALL statements allow only a restricted form of conditional assignments via the predefined function SELECT. The point is, whether this offers sufficient flexibility for more complicated applications. However, more powerful constructs do require either a significantly more complicated translation or cannot be tailored to vector instructions at all.

7.3 Conclusions

Much effort spent today on advanced compiler technology goes into the development of techniques aiming at the improvement of the execution speed of generated code. These techniques may be roughly divided into two categories: a) *machine-specific* techniques for the better use of limited resources (such as registers or functional units) of a particular target machine, and b) *machine-independent* analyzing techniques to reveal inherent properties of a particular program (such as independence of statements), which are used later to automatically rewrite the program so that it can be mapped on special machine architectures. In general, *machine specific* restrictions cannot be circumvented by suitable coding or by choosing a better programming language (it is neither desirable nor reasonably possible to ask the programmer to perform instruction scheduling). Thus, the classical optimization techniques of category a) are justified if a reasonable ratio between cost and benefit is achieved. Techniques of category b) stand in strong contrast to category a): a compiler tries here to reveal program properties that the programmer has been even aware of (i.e. the compiler does reverse-engineering). If taking advantage of such properties means a benefit in an order of magnitude, a programming language must provide a means to explicitly specify this property. We would like to quote here C. A. R. Hoare, who wrote 20 years ago in a similar context: "What a pity that the designers of these languages take such trouble to give such trouble to their users and to themselves." [Hoare 1974].

If we replace an ordinary loop by an ALL statement without changing the effect of the program, we can observe two points: first of all, the program becomes simpler because unnecessary specifications (such as an iteration order) are removed. If it was a WHILE loop that had been replaced, the decrease of complexity is immediately noticable: termination of the loop is not a question any more. Secondly, as an immediate consequence of the first point, a compiler has more freedom in the translation of the statement as it must observe fewer constraints. Thus, it may use this freedom to generate better code but it is not forced to do so, which is an important aspect. We have a similar situation with Dijkstra's guarded commands: if several guards evaluate to true, it is not determined which statement sequence with a valid guard is going to be executed. Again, the absence of an explicitly specified order simplifies reasoning about the program. A compiler accepting guarded commands may take advantage of this additional freedom to generate more efficient code, but it may also use the source text order to ease code generation. Future programming languages should better exploit this relationship.

Bibliography

Abrams, P. S. [1970]. *An APL Machine*, PhD thesis, Computer Science Departement, Stanford University, TR STAN–CS–70–158. Proposes a two-processor implementation for an optimized execution of APL using mathematical properties of APL expressions.

Aho, A. V. and J. D. Ullman [1977]. *Principles of Compiler Design*, Addison-Wesley.

Allen, R. and K. Kennedy [1987], "Automatic Translation of FORTRAN Programs to Vector Form", *ACM Transactions on Programming Languages and Systems*, 9:4 (October), 491–542.

Auslander, M. and M. Hopkins [1982]. "An Overview of the PL.8 Compiler", *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, Boston (June), 22–31. Instruction scheduling is performed before final register allocation in the PL.8 compiler.

Banerjee, U. [1988]. *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, 1988. Gives a thourough introduction into the topic.

Basili, V. R. and J. C. Knight [1975]. "A Language for vector machines", *Proceedings of a Conference on Programming Languages and Compilers for Parallel and Vector Machines*, ACM Sigplan 10:3 (March). Describes the programming language SL/1 for CDC STAR-100 machines.

Berry, P. [1979]. *Sharp APL Reference Manual*, I. P. Sharp Associates.

Bradlee, D. G., S. J. Eggers and R. R. Henry [1991]. "Integrating Register Allocation and Instruction Scheduling for RISCs", *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, (April), 122–131.

Brainerd, W. (ed.) [1978]. "Fortran 77", *Communications of the ACM*, 21:10 (October), 806–820.

Brandis, M., R. Crelier, M. Franz and J. Templ [1992]. *The Oberon System Family*, Technical Report 174, Institut für Computersysteme, ETH Zürich.

Brandis, M. and H. Mössenböck [1993]. "Single-Pass Generation of Static Single Assignment Form for Structured Languages", submitted to the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) 1994*. Not "well-structured" statements (such as loop and corresponding exit statements) may introduce significant burdens for optimizing compilers using static single assignment form.

Burke, M. and R. Cytron [1986]. "Interprocedural Dependence Analysis and Parallelization", *ACM SIGPLAN Symposium on Compiler Construction*, 162–174.

Cardelli, L., J. Donahue, L. Glassman, M. Jordan, B. Kalsow and G. Nelson [1988]. *Modula–3 Report (revised)*, DEC SRC Research Report 52, DEC Systems Research Center, Palo Alto, CA (November).

Cardelli, L. [1989]. *Typeful Programming*, DEC SRC Research Report 45, DEC Systems Research Center, Palo Alto, CA (May). Gives a list of relevant criteria on the "usefulness" of a language.

Chaitin, G. J. [1982]. "Register Allocation and Spilling via Graph Coloring", *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, Boston (June), 98–105.

Chandy, K. M. and J. Misra [1988]. *Parallel Program Design - A Foundation*, Addison-Wesley, Inc.

Cocke, J. and J. T. Schwartz [1970]. *Programming Languages and Their Compilers – Preliminary Notes*, Second Revised Version, Courant Institute of Mathematical Sciences, New York University, 320–334. Describes value numbering for basic block optimization.

Cohen, N. H. [1991]. "Type-Extension Type Tests Can Be Performed In Constant Time", *ACM Transactions on Programming Languages and Systems*, 13:4 (October), 626–629.

Crelier, R. [1990]. OP2: *A Portable Oberon Compiler*, Technical Report 125, Institut für Computersysteme, ETH Zürich.

Crelier, R. [1993]. *Separate Compilation and Module Extension*, PhD Thesis, to be published, ETH Zürich.

Cray [1986]. *Fortran (CFT) Reference Manual*, Cray Research, Inc., Publication SR–0009.

Cray [1988]. *CRAY Y-MP Computer Systems Functional Description Manual*, Cray Research, Inc., Publication HR–4001.

Davidson, J. W. and C. W. Fraser [1984]. "Register Allocation and Exhaustive Peephole Optimization", *Software – Practice and Experience*, 14:9 (September).

Davidson, J. W. [1986]. "A Retargetable Instruction Reorganizer", *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, (June), 234–241.

Dijkstra, E. W. [1976]. *A Discipline of Programming, Prentice Hall*, Englewood Cliffs.

Dongarra, J. J., C. B. Moler, J. R. Bunch and G. W. Stewart [1979]. *LINPACK – User's Guide*, SIAM, Philadelphia.

Erickson, D. B. [1975]. "Array Processing on an Array Processor", *Proceedings of a Conference on Programming Languages and Compilers for Parallel and Vector Machines*, ACM Sigplan 10:3 (March). Describes the Fortran dialect IVTRAN for the ILLIAC IV computer.

Freiburghouse, R. A. [1974]. "Register Allocation Via Usage Counts", *Communications of the ACM*, 17:11 (November), 638–642.

Gibbons, P. B. and S. S. Muchnick [1986]. "Efficient Instruction Scheduling for a Pipelined Architecture", *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto (June).

Gries, D. [1981]. *The Science of Programming*. Texts and Monographs in Computer Science, Springer.

Griesemer, R. [1991]. *On the Linearization of Graphs and Writing Symbol Files*, Technical Report 156, Institut für Computersysteme, ETH Zürich. Describes the symbol file generation method used in the Oberon-V compiler.

Griesemer, R. [1992]. "Scheduling Instructions by Direct Placement", *Proceedings of the 4th International Conference on Compiler Construction*. Lecture Notes in Computer Science 641, Springer, 229–235. Describes the instruction scheduling method used in the Oberon-V compiler.

Gross, T. [1983]. *Code Optimizations of Pipeline Constraints*, Technical Report 83–255, Computer Systems Laboratory, Stanford University. Gives a survey of instruction scheduling and presents two scheduling approaches for MIPS.

Gutknecht, J. [1985]. *Compilation of Data Structures: A New Approach to Efficient Modula-2 Symbol Files*, Technical Report 64, Institut für Computersysteme, ETH Zürich.

Heeb, B. [1988]. *Design of the Processor-Board for the Ceres-2 Workstation*, Technical Report 93, Institut für Computersysteme, ETH Zürich.

Hennessy, J. L. and T. Gross [1983]. "Postpass Code Optimization of Pipeline Constraints", *ACM Transactions on Programming Languages and Systems*, 5:3 (July), 422–448. A revised version of [Gross 1983].

Hennessy, J. L. and D. A. Patterson [1990]. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc.

Hoare, C. A. R. [1969]. "An axiomatic basis for computer programming", *Communications of the ACM*, 12:10 (October), 576–580, 583.

Hoare, C. A. R. and N. Wirth [1973]. "An axiomatic definition of the programming language Pascal", *Acta Informatica*, 2:4, 335–355.

Hoare, C. A. R. [1974]. "Hints on programming-language design", *Computer Systems Reliability*, State of the Art Report 20, 505–534.

Iverson, K. E. [1962]. *A Programming Language*, John Wiley and Sons, New York. Introduces APL as a concise mathematical notation.

Kane, G. [1987]. *MIPS R2000 RISC Architecture*, Prentice Hall. A well-known pipelined processor architecture.

Lamport, L. [1974]. "The Parallel Execution of DO Loops", *Communications of the ACM*, 17:2 (February), 83–93.

Lampson, B. W., J. J. Horning, R. L. London, J. G. Mitchell and G. J. Popek [1977]. "Report On The Programming Language Euclid", *ACM SIGPLAN Notices*, 12:2 (February). No side effects are allowed in Euclid functions.

Landskov, D., S. Davidson, B. Shriver and P. W. Mallett [1980]. "Local Microcode Compaction Techniques", *ACM Computing Surveys*, 12:3 (September), 261–294. Some ideas (list scheduling) are used in an appropriate form for instruction scheduling.

Lawrie D. H., T. Layman, D. Baer and J. M. Randal [1975]. "Glypnir – A programming Language for Illiac IV", *Communications of the ACM*, 18:3 (March).

Lawson, C., R. Hanson, D. Kincaid and F. Krogh [1979]. "Basic Linear Algebra Subprograms for Fortran Usage", *ACM Transactions on Mathematical Software*, Vol. 5.

MathWorks [1987]. *PRO–MATLAB for Sun Workstations*, User Guide, The MathWorks, Inc.

Metcalf, M. and J. Reid [1987]. *Fortran 8x Explained*, Oxford University Press. Explaines what now is called Fortran-90.

Millstein, R. E. and C. A. Muntz [1975]. "The ILLIAC IV Fortran Compiler", *Proceedings of a Conference on Programming Languages and Compilers for Parallel and Vector Machines*, ACM Sigplan 10:3 (March).

Mössenböck, H. and N. Wirth [1991]. "The Programming Language Oberon-2", *Structured Programming*, 12:4.

Naur, P. (ed.) [1960]. *Report on the Algorithmic Language Algol 60*, Technical Report, Regnecentralen, Copenhagen.

NEC [1989]. *NEC Supercomputer SX-3 Series – General Description*, NEC Corporation.

Perrott, R. H. [1979]. "A Language for Array and Vector Processors", *ACM Transactions on Programming Languages and Systems*, 1:2, 177–195. Describes the programming language Actus, an offspring of Pascal.

Perrott, R. H., D. Crookes and P. Milligan [1983]. "The Programming Language ACTUS", *Software – Practice and Experience*, Vol. 13, 305–322.

Reiser, M. and N. Wirth [1992]. *Programming in Oberon – Steps beyond Pascal and Modula*, Addison Wesley.

Robbins, K. A. and S. Robbins [1987]. *The Cray X-MP/Model 24 – A Case Study in Pipelined Architecture and Vector Processing*, Lecture Notes in Computer Science 374, Springer.

Russel, R. M. [1978]. "The CRAY-1 Computer System", *Communications of the ACM*, 21:1 (January), 63–72.

Stevens Jr., K. G. [1975]. "CFD – A Fortran–like Language for the ILLIAC IV", *Proceedings of a Conference on Programming Languages and Compilers for Parallel and Vector Machines*, ACM Sigplan 10:3 (March). Describes the programming language CFD for the ILLIAC IV computer.

Sun [1987]. *The SPARC Architecture Manual*, Sun Microsystems, Inc. A well-known pipelined processor architecture.

Templ, J. [1989]. Private communication. J. Templ implemented the first version of the symbol file browser for the Oberon System.

Warren, H. S., Jr. [1990]. "Instruction Scheduling for the IBM RISC System/6000 Processor", *IBM Journal of Research and Development*, 34:1 (January), 85–92.

Winston, P. H. and B. K. P. Horn [1984]. *Lisp*, 2nd edition, Addison-Wesley. Contains a definition of the notion "side effect".

Wirth, N. [1971]. "The Programming Language Pascal", *Acta Informatica*, 1, 35–63.

Wirth, N. [1972]. *On "Pascal", Code Generation, and the CDC 6000 Computer*, Technical Report STAN-CS-72-257, Computer Science Departement, Stanford University.

Wirth, N. [1985a]. *Programming in Modula-2*, Springer, third corrected edition.

Wirth, N. [1985b]. *A Fast and Compact Compiler for Modula-2*, Technical Report 64, Institut für Computersysteme, ETH Zürich.

Wirth, N. [1988a]. "The Programming Language Oberon", *Software – Practice and Experience*, 18:7 (July), 671–690.

Wirth N. and J. Gutknecht [1988b]. "The Oberon System", *Software – Practice and Experience*, 19:9 (Sept.).

Wirth, N. [1990]. *From Modula to Oberon*, Technical Report 143, Institut für Computersysteme, ETH Zürich.

Wirth, N. and J. Gutknecht [1992]. *Project Oberon – The Design of an Operating System and Compiler*, Addison Wesley, New York, 332–455.

Zima, H., P. Brezany, B. Chapman, P. Mehrotra and A. Schwald [1992]. *Vienna Fortran – A Language Specification* (Version 1.1), ICASE Interim Report 21, NASA Langley Research Center, Hampton, Virginia, NASA Contractor Report 189629.

Appendix A: The Programming Language Oberon-V

1 Introduction

Oberon-V is a general-purpose programming language in the tradition of the Algol family. It evolved from Oberon and provides mainly the same capabilities; however, Oberon-V is simpler in some aspects. Its principal new features are the possibility to express parallelism in assignments and a construct to specify subarrays. The former allows for efficient translation into vector instructions (which also explains the name of the language), while the latter is an essential prerequisite for the programming of numerical applications. Furthermore, the concept of the side effect free function has been added.

This report defines Oberon-V. It serves as a reference manual for programmers and compiler writers but is not intended as a tutorial. What remains unspecified is mostly left so intentionally. Within the (informal) definition of new terms, references to terms defined elsewhere are enclosed in brackets [1].

2 Syntax

A language is a set of finite sentences, namely the sentences that are well-formed according to its defining grammar. In Oberon-V these sentences are called *compilation units*. Each unit is a sequence of words (lexical entities) from a vocabulary. The lexical entities are called *symbols* in Oberon-V. They are composed of sequences of characters.

The syntax of a language is defined by a set of *productions*; i.e. rules which specify how a syntactic entity is composed of other entities. The syntax of

Oberon-V is described using an Extended Backus-Naur Formalism (EBNF): brackets [and] denote optionality of the enclosed sentential form, braces { and } denote its repetition (possibly zero times). Alternative forms are separated by vertical bars |. Syntactic entities (non-terminal symbols) are denoted by English words starting with a capital letter (e.g. Statement). Symbols of the language vocabulary (terminal symbols) are either denoted by English words starting with a small letter (e.g. number), or are written completely in capital letters (e.g. BEGIN), or are denoted by strings enclosed in double quotes (e.g. ":="). Productions are marked by a bold vertical bar at the left margin of a line.

3 Vocabulary and Representation

Each symbol is a well-formed sequence of characters from an alphabet which is a subset of the ASCII set. The following lexical rules must be observed: any number of blanks, new-line characters and comments can appear between two symbols; they are required only when essential for the separation of two consecutive symbols. A comment is any sequence of characters enclosed in (* and *) brackets and may be nested. Comments do not affect the meaning of a program. Capital and small letters are considered as being distinct. There are the following classes of symbols:

3.1 Identifiers

Identifiers are sequences of letters and digits. The first character must be a letter. Two identifiers are *equal* if they consist of the same sequence of characters.

```
ident  =  letter {letter | digit}.
letter  =  "A" ... "Z" | "a" ... "z".
digit  =  "0" ... "9".
```

Examples:

```
pi   x1   Node   Matrix   WriteReal
```

3.2 *Numbers*

Numbers are either integer, real or imaginary numbers. They are always unsigned. Integers are sequences of digits and may be followed by a suffix letter. If no suffix is specified, the representation is decimal. The suffix H indicates hexadecimal representation. Real numbers are sequences of digits containing a decimal point. Optionally they may also contain a decimal scale factor. The letter E is then pronounced as "times ten to the power of". Imaginary numbers are denoted by real numbers followed by the suffix letter i which is pronounced as "multiplied by the imaginary number i" (with i∗i = –1). They are of type COMPLEX.

```
number = integer | real ["i"].
integer = digit {digit} | digit {hexDigit} "H".
real = digit {digit} "." {digit} [scaleFactor].
scaleFactor = "E" ["–"] digit {digit}.
```

Examples:

Number	*Value*	*Type*
1993	1993	INTEGER
0F7H	247	INTEGER
17.4	17.4	REAL
9.806665E–6	0.000009806665	REAL
4.5E2i	0.0 + 450.0 ∗ 1.0i	COMPLEX

3.3 *Character Constants*

Character constants are either denoted by enclosing a character in single quotes (') or by the ordinal number of the character in hexadecimal notation followed by the letter X. The enclosed character must be in the range " " ... "~" but must not be a single quote.

```
character = "'" char "'" | digit {hexDigit} "X".
char = " " ... "~".
```

Examples:

```
'A'   'z'   0X   27X   '~'
```

3.4 *Strings*

Strings are sequences of characters enclosed in double quotes (") which cannot occur within the string. A string is always terminated by an (invisible) 0X character. The number of characters (including the terminating 0X character) is called the *length* of the string. The length of the *empty string* "" is 1.

```
string  =  """ {char} """.
```

Examples:

```
"Oberon-V"   "Don't panic!"   ""
```

3.5 *Operators and Delimiters*

Operators and delimiters are the special characters, character pairs or *reserved words* listed below. These reserved words consist exclusively of capital letters and cannot be used in the role of identifiers.

+	=	#	ALL	END	RECORD
−	<	>=	ARRAY	IF	REPEAT
*	>	<=	BEGIN	IN	RETURN
/	(	)	CASE	IS	THEN
&	[	]	CONST	MOD	TYPE
|	{	}	DIV	MODULE	UNTIL
~	:	:=	DO	NIL	VAR
%	,	;	ELSE	OF	WHILE
↑	.	..	ELSIF	PROCEDURE	

4 Declarations and Scope Rules

A declaration introduces an identifier and associates it with an *object*. Objects are either constants, types, variables, record fields, procedures or ranges. After its introduction the identifier denotes the associated object. The program text where this association is valid is called the *scope* of the identifier. The scope extends textually from the introduction of the identifier to the end of the (module or procedure) *block* to which the declaration belongs – and hence to which the identifier is *local* – but excludes the scopes of equal identifiers declared in nested blocks. A block comprises the (module or procedure) *heading* and the succeeding *body* [8][12]. Within a given scope, no identifier must be declared twice. The following exceptions complete the scope rules:

1. Field identifiers of a record declaration [6.3] are valid in field designators [10.3] only.
2. An identifier T used within a pointer type [6.4] of the form ↑T can be declared textually following the use of the pointer type but must lie within the same block in which ↑T is used.
3. The scope of range identifiers [9] extends textually from the introduction of the identifier to the end of the enclosing array constructor [10.6] or ALL statement [11.9].

An identifier declared local to a module block may be followed by an export mark (*) in its declaration. The export mark indicates that the identifier and hence the associated object is *exported*. In this case, the identifier may be used in other modules, if they *import* the declaring module [12]. The identifier is then prefixed by the identifier denoting its module. The prefix and the identifier are separated by a period and together are called a *qualified identifier*. Two qualified identifiers are *equal* if both the module prefix and the identifier denoting the imported object are equal [3.1].

```
IdentDef  =  ident ["*"].
QualIdent  =  [ident "."] ident.
```

The following identifiers are predefined; they may be thought of beeing declared in an additional scope enclosing all other scopes. Their meaning is defined in the indicated sections.

ABS	[8.2]	EXCL	[8.2]	NEW	[6.4]
ASL	[8.2]	FALSE	[6.1]	ODD	[8.2]
ASR	[8.2]	FLOOR	[8.2]	ORD	[8.2]
ASSERT	[8.2]	HALT	[8.2]	PROD	[8.2]
BOOLEAN	[6.1]	IM	[8.2]	RE	[8.2]
CEILING	[8.2]	INC	[8.2]	REAL	[6.1]
CHAR	[6.1]	INCL	[8.2]	SELECT	[8.2]
CHR	[8.2]	INTEGER	[6.1]	SET	[6.1]
COMPLEX	[6.1]	LEN	[8.2]	SUM	[8.2]
DEC	[8.2]	MAX	[8.2]	TRUE	[6.1]
EPS	[8.2]	MIN	[8.2]	TRUNC	[8.2]

5 Constant Declarations

A constant declaration introduces an identifier and associates it with the value of a *constant expression*. A constant expression is any expression whose operands consist only of literals, set constants, identifiers denoting constants, the predefined identifiers TRUE and FALSE as well as applications of predefined functions to constant expressions.

```
ConstantDeclaration  =  IdentDef "=" ConstExpression.
ConstExpression  =  Expression.
```

Examples:

```
M = 8
N = M*M
CR = 0DX
pi = 3.14159265
all = /{}
```

6 Type Declarations

A data type determines the set of values which variables of that type may assume, and the operators that are applicable. A type declaration associates a (type) identifier with a type. There are *unstructured* and *structured* types. Unstructured types are either *basic types* denoted by predefined identifiers, pointer or procedure types. Structured types, namely arrays and records, define also the structure of variables of this type, i.e. the types and possibly the names of their *components*.

```
TypeDeclaration  =  IdentDef "=" Type.
Type  =  QualIdent | ArrayType | RecordType | PointerType | ProcedureType.
```

If two (possibly qualified) type identifiers x and y are *equal* [4] or if they are declared to denote equal types in a type declaration of the form x = y, then the types T(x) and T(y) denoted by them are *equal*.

Examples:

```
Ident = ARRAY 16 OF CHAR
RealVector = ARRAY OF REAL
Matrix = ARRAY 100, 100 OF COMPLEX
List = RECORD END
Pair = RECORD (List)
  car, cdr: ↑List
END
Symbol = RECORD
  mark: INTEGER;
  left, right: ↑Symbol;
  name: ARRAY 32 OF CHAR;
  bound: ↑List
END
Function = PROCEDURE (x: REAL): REAL
```

6.1 Basic Types

There are 6 basic types in Oberon-V. The corresponding predefined identifiers and the values comprised by its associated types are the following:

BOOLEAN	the truth values TRUE and FALSE
CHAR	all characters between 0 and MAX(CHAR)
SET	all sets containing elements between 0 and MAX(SET)
INTEGER	all integers between MIN(INTEGER) and MAX(INTEGER)
REAL	all real numbers between MIN(REAL) and MAX(REAL)
COMPLEX	all complex numbers x + y * 1.0i with REAL x and y

The last three types are called *numeric types* and they form a type hierarchy; i.e. values of the smaller type are included by the larger type.

$$\text{INTEGER} \subset \text{REAL} \subset \text{COMPLEX}$$

6.2 *Array Types*

An array is a structured type composed of a fixed number of *elements* which are all of the same type, called the *element type*. The element type must not be a structured type which is or contains the declared array itself. The number of elements of an array is called its *length*. The elements are designated by indices which are integers between 0 and the length minus 1. The number of *dimensions* of an array is the number of dimensions of the element type plus 1. A type which is not an array has zero dimensions.

```
ArrayType  =  ARRAY [Length {"," Length}] OF Type.
Length  =  ConstExpression.
```

A declaration of an (n-dimensional) array of the form

ARRAY L_1, L_2 ... L_n OF T

is an abbreviation of the declaration

```
ARRAY L1 OF
  ARRAY L2 OF
    ...
      ARRAY Ln OF T
```

Arrays declared without length are called *open arrays*. An open array must not be a component of a type which is not an open array.

Examples:

```
ARRAY OF CHAR
ARRAY N–1 OF INTEGER
ARRAY 100, 100 OF COMPLEX
```

6.3 Record Types

A record is a structured type composed of a fixed number of components of possibly different types. The record type definition specifies for each component, called *field*, its type and an identifier which denotes the field. Within a record declaration, no field identifier must be declared twice. The type of a field must not be a structured type which is or contains the declared record itself. If a record type is exported [4], field identifiers that are to be visible outside the declaring module must be marked too. They are called *public fields*; the unmarked components are called *private fields*.

```
RecordType  =  RECORD ["(" BaseType ")"] [FieldList {";" FieldList}] END.
BaseType  =  QualIdent.
FieldList  =  IdentList ":" Type.
IdentList  =  IdentDef {"," IdentDef}.
```

Record types are extensible, i.e. a record type T1 can be declared as a *direct extension* of another record type T which is then called its *direct base type*. The extended type T1 consists of the fields of its base type and of the fields which are declared in T1. A field identifier declared in a (direct) extension must be different from any identifier declared in any of the base type(s) of the record.

```
T = RECORD ... END
T1 = RECORD (T) ... END
```

A record type T1 *extends* a type T, if it is equal to T or if it directly extends an extension of T. Conversely, a record type T is a *base type* of T1, if it is equal to T1 or if it is a direct base type of a base type of T1.

Examples:

```
RECORD
  hour, min, sec: INTEGER
END
RECORD
  name: ARRAY 32 OF CHAR;
  exported: BOOLEAN;
  mode: INTEGER;
  value: REAL
END
```

6.4 Pointer Types

The pointer type P = ↑T comprises the set of all pointers to variables of type T, and it includes the value NIL which points to no variable at all. The pointer type is said to be *bound* to T, and T is called the *pointer base type* of P. T must be a structured type (i.e. an array or a record type). Two pointer types are *equal*, if their base types are equal.

```
PointerType = "↑" Type.
```

If p designates a variable of type ↑T, where T is not an open array type, then a call of the predefined procedure NEW(p) allocates a new variable of type T and the pointer to it is assigned to p. If p designates a pointer variable bound to an n-dimensional open array, then the call NEW(p, e_1, e_2, ... e_n) allocates a new n-dimensional open array variable with lengths given by the values of the expressions e_1, e_2, ... e_n, and the pointer to this array is assigned to p. A variable allocated by NEW is not local to any block.

6.5 Procedure Types

A procedure type P comprises the set of all procedures whose types are equal to P and it includes the value NIL which denotes no procedure at all.

Two procedures have *equal types*, if all corresponding parameters and the result types, if any, are equal. Two parameters are *equal* if they are of the same kind [8.1], if they are denoted by equal identifiers and if they have equal types.

```
ProcedureType = PROCEDURE [ParameterList].
```

Examples (see also examples in [6]):

```
PROCEDURE
PROCEDURE (p: ↑List)
PROCEDURE (x: COMPLEX): REAL
PROCEDURE (VAR a: ARRAY OF INTEGER; scale: REAL)
PROCEDURE (p: ↑Symbol; cmp: PROCEDURE (x, y: REAL): INTEGER)
```

7 Variable Declarations

Variable declarations introduce new variables by declaring an identifier and a type for them. All variables whose identifiers appear in the same identifier list have equal types. The type of a variable (which is not a parameter) must not be an open array [6.2]. A *structured variable* is a variable which has a structured type [6]. It consists of *component variables* (i.e. array elements or record fields) as specified by its type.

```
VariableDeclaration = IdentList ":" Type.
```

Record variables have both a *static* and a *dynamic type*. The former is the type with which they are declared and which is simply called their type; the latter is the type which their values may assume at run-time. Variable parameters of record type and record variables referred to by pointers may have a dynamic type which is an extension of their static type.

Examples (see also examples in [5] and [6]):

```
ch: CHAR
i, j, k: INTEGER
x, y: REAL
u, v: COMPLEX
ok: BOOLEAN
s: SET
f: Function
A: Matrix
p, q: ↑Pair
root: ↑Symbol
index: ARRAY 100 OF INTEGER
a, b, c: ARRAY N OF REAL
id: Ident
```

8 Procedure Declarations

A procedure declaration consists of two parts, a *procedure heading* and a *body*. The heading specifies the procedure identifier and the procedure type (determined by the parameters and the result type, if any). The body contains declarations and statements that are executed, when the procedure is called.

The procedure identifier must be repeated at the end of the procedure declaration.

```
ProcedureDeclaration = ProcedureHeading ";" Body END ident.
ProcedureHeading = PROCEDURE IdentDef [ParameterList].
ForwardDeclaration = PROCEDURE "↑" IdentDef [ParameterList].
Body = DeclarationSequence [BEGIN StatementSequence].
DeclarationSequence =
  [CONST {ConstDeclaration ";"}]
  [TYPE {TypeDeclaration ";"}]
  [VAR {VarDeclaration ";"}]
  {ProcedureDeclaration ";" | ForwardDeclaration ";"} .
```

All objects declared in a procedure body are *local* to it [4]. Values of local variables are undefined upon entry to a procedure. Objects declared in the environment of the procedure are also visible within the procedure, with the exception of those objects that have equal identifiers as locally declared objects. The use of the procedure identifier in a call within its declaration implies recursive activation of the procedure.

Procedures are either *proper procedures* or *function procedures*. The former represent an (abstract) action while the latter compute a value. The execution of a function procedure must terminate with an explicit RETURN statement [11.9] which also denotes the value to be returned. The result type cannot be a structured type. Function procedures must produce no *side effects*; i.e. within a function procedure the following restrictions apply:

1. no value must be assigned to a variable which is not local, and
2. no proper procedure (except predefined ones) must be called.

A procedure identifier may be introduced by a *forward declaration*. In this case, the scope of the procedure identifier extends from its introduction to the end of the block to which the forward declaration belongs. The actual declaration containing the procedure body must follow later within the same block. The procedure types of both declarations must be equal [6.5].

8.1 Parameters

Parameters are identifiers declared in a parameter section of a procedure heading. They are *local* to the procedure block and they denote the arguments specified in the procedure call. The association between parameters and arguments is established, when the procedure is called.

There are two kinds of parameters, namely *value* and *variable parameters*, indicated in the parameter section by the absence or presence of the reserved word VAR. Value parameters denote *local variables* to which the value of the associated argument is assigned as initial value. Variable parameters are associated with arguments that must be variables and they denote these *non-local variables* (i.e. a variable parameter is a local identifier for a non-local variable).

```
ParameterList  =  "(" [ParameterSection {";" ParameterSection}] ")" [":" Type].
ParameterSection  =  [VAR] ident {"," ident} ":" Type.
```

The types of the parameters (i.e. the types of the variables denoted by the parameters) are specified in the parameter sections; all parameters which appear in the same parameter section are of the same parameter kind and have equal types. The type of value parameters cannot be a structured type.

Examples of procedure declarations (see also examples of [6]):

```
PROCEDURE ReadInt (VAR x: INTEGER);
  VAR n, s: INTEGER; ch: CHAR;
BEGIN s := 0; n := 0; Read(ch);
  WHILE ch < 80X DO INC(n, ASL(ORD(ch), s)); INC(s, 7); Read(ch) END;
  x := n + ASL(ORD(ch) – 192, s)
END ReadInt

PROCEDURE WriteInt (x: INTEGER);
BEGIN
  WHILE (x < –64) OR (x > 63) DO
    Write(CHR(x MOD 128)); x := x DIV 128
  END;
  Write(R, CHR(x + 192))
END WriteInt
```

```
PROCEDURE saxpy (a: REAL; VAR x, y: RealVector);
BEGIN ALL i = 0 .. LEN(x)–1 DO y[i] := a ∗ x[i] + y[i] END
END saxpy

PROCEDURE Size (p: ↑Symbol): INTEGER;
BEGIN
  IF p = NIL THEN RETURN 0
  ELSE RETURN 1 + Size(p.left) + Size(p.right)
  END
END Size
```

8.2 Predefined Procedures

The following tables list the predefined procedures. Some of them are *generic*, i.e. they apply to several types of arguments. x, y and n stand for expressions, v stands for a variable and T for a type.

Function procedures:

Call	*Argument types*	*Result type*	*Function*
ABS(x)	INTEGER	INTEGER	absolute value of x
	REAL	REAL	
	COMPLEX	REAL	
ASL(x, n)	x, n: INTEGER	INTEGER	$x * 2^n$ $(n \geq 0)$
ASR(x, n)	x, n: INTEGER	INTEGER	x DIV 2^n $(n \geq 0)$
ODD(x)	INTEGER	BOOLEAN	x MOD 2 = 1
ORD(x)	CHAR	INTEGER	ordinal number of x
CHR(x)	INTEGER	CHAR	x = CHR(ORD(x))
EPS(x)	REAL	REAL	small. real ε: x + ε > x
TRUNC(x)	REAL	INTEGER	integer part of x
FLOOR(x)	REAL	INTEGER	largest integer i with i ≤ x
CEILING(x)	REAL	INTEGER	small. integer i with i ≥ x
RE(x)	COMPLEX	REAL	real part of x
IM(x)	COMPLEX	REAL	imaginary part of x
LEN(v, n)	v: array	INTEGER	length of v in dim. n
	n: INTEGER		
LEN(v)	v: array	INTEGER	LEN(v) = LEN(v, 0)
MIN(T)	CHAR	CHAR	0X

	SET	INTEGER	0
	INTEGER	INTEGER	smallest integer
	REAL	REAL	smallest real number
MAX(T)	CHAR	CHAR	largest character
	SET	INTEGER	largest set element
	INTEGER	INTEGER	largest integer
	REAL	REAL	largest real number
SUM(x)	n-dim. array of T T = numeric type	T	sum of all elements of v
PROD(x)	n-dim. array of T T = numeric type	T	prod. of all elements of v
SELECT(n, x, y)	n: BOOLEAN x, y: T T = unstruct. type	T	if n then x else y (both arguments are evaluated always)

Proper procedures:

Call	*Argument types*	*Function*
INC(v)	INTEGER	v := v + 1
DEC(v)	INTEGER	v := v – 1
INCL(v, x)	v: SET; x: INTEGER	v := v + {x}
EXCL(v, x)	v: SET; x: INTEGER	v := v / {x}
NEW(v)	pointer type	allocate v↑
NEW(v, x_1, ... x_n)	v: ↑ n-dim. open array x_1, ... x_n: INTEGER	allocate array v↑ with lengths x_1, ... x_n
HALT(n)	integer constant	terminate program execution
ASSERT(n)	BOOLEAN	terminate program execution if ~n

The interpretation of the argument x in HALT(x) is left to the underlying operating system.

9 Range Declarations

Range declarations occur within array constructors [10.6] and ALL statements [11.9] only. A range declaration introduces a range identifier and associates it with a *range* a .. b, specified by two integer expressions a and b that must not contain any range identifiers themselves. A range a .. b denotes the set of integers between a and b (inclusive). The number of integers denoted by a range is called its *length*. A range declaration that introduces *n* range identifiers is called *n-dimensional*. A range identifier may be used within an expression [10.3].

```
RangeDeclaration = RangeList {"," RangeList}.
RangeList = ident {"," ident} "=" Range.
Range = Expression ".." Expression.
```

An (n-dimensional) range declaration of the form

$r_1, r_2, \ldots r_n = a \mathinner{..} b$

is an abbreviation of the declaration

$r_1 = a \mathinner{..} b, r_2 = a \mathinner{..} b, \ldots r_n = a \mathinner{..} b$

Examples:

```
r = M .. N
i, j, k = 0 .. LEN(A)–1
i = 1 .. M, j = 1 .. N
i, j = –10 .. 10
```

10 Expressions

An expression specifies the computation of a *value*; it consists of *operands* and *operators*. The value of the expression is determined by applying the operators according to their precedences to the values of the operands. Operands are either literals, sets, designators, function calls or expressions.

10.1 Literals

Literals are numbers, character constants, strings and the value NIL. The value of a literal is the value it represents.

```
Literal = number | character | string | NIL.
```

10.2 Sets

A set represents a value of type SET by listing its elements between braces. The elements must be integer values. The notation a .. b is an abbreviation for the elements a, a+1, .. b. A *set constant* is a set where all elements are denoted by constant expressions.

```
Set = "{" [Element {"," Element}] "}".
Element = Expression [".." Expression].
```

Examples:

```
{0, 4, 10}   {i}   {0 .. i}   {i .. i+4, 42}
```

10.3 Designators

A designator is a qualified identifier [4] possibly followed by selectors or type guards. The value of a designator denoting a constant is the value of the constant. The value of a designator denoting a type or procedure is the denoted type or procedure. A designator denoting a variable stands for this variable. Within an array constructor [10.6] or an ALL statement [11.9], a range identifier denotes any integer value within the associated range.

```
Designator = QualIdent {"[" ExpressionList "]" | "." | "↑" | "{" QualIdent "}"}.
ExpressionList = Expression {"," Expression}.
```

If A designates an array variable, then A[e] denotes the element of A whose index is the current value of the expression e. A designator of the form A[e_1][e_2] ... [e_n] may be abbreviated to A[e_1, e_2, ... e_n]. IF p designates a pointer variable, then p↑ denotes the variable which is referenced by p. If r designates a record variable, then r.f denotes the field f of r. Designators of the forms p↑.f and p↑[e] may be abbreviated to p.f and p[e] respectively.

If the variable v is of record type [6.3], the *type guard* v{T} asserts that the dynamic type of v is an extension of T. If v is a pointer type [6.4], v{T} asserts that the dynamic type of v↑ is an extension of T. The static type of the designator v{T} is then T (or ↑T respectively). If the assertion fails, the program execution is aborted. The type guard is applicable, if

1. v is a record variable parameter or v is a pointer (v ≠ NIL), and if
2. the record type T is an extension of the static type of v (or v↑ respectively).

Examples (see also examples in [7]):

Designator	*Type*
i	INTEGER
A[i, 10]	COMPLEX
p.car↑	List
q.cdr{Pair}	↑Pair
root.name[k]	CHAR
f	Function

10.4 Function Calls

A function call activates a function procedure [8]. The value of a function call is the value returned by the function procedure. The call consists of a designator denoting a function procedure followed by a possibly empty list of arguments [10.5].

```
FunctionCall = Designator "(" [ArgumentList] ")".
```

Examples (see also examples in [7]):

```
f(x*pi)
Size(root)
TRUNC(x*x + y*y)
```

10.5 Arguments

Within a procedure, the arguments are denoted by their corresponding parameters specified in the procedure declaration. The correspondence is determined by the positions of the arguments and parameters within the argument and parameter list.

```
ArgumentList = Argument {"," Argument}.
Argument = Expression | ArrayConstructor .
```

There are two kinds of parameters, namely value and variable parameters [8.1]. In case of a value parameter, the corresponding argument must be an expression. This expression is evaluated prior to the procedure call and the resulting value is assigned to the value parameter. Argument and parameter must observe the rules for assignments.

In case of a variable parameter, the corresponding argument must be a designator or an array constructor [10.6] both denoting a variable, or a string (since value parameters cannot have a structured type [8.1]). The parameter and the argument type must be equal, except for record and open array parameters, where the following rules apply:

1. If the type of the parameter is a record [6.3], the type of the argument must be an extension of the parameter's type. After procedure activation, the dynamic type of the parameter corresponds to the dynamic type of the argument.

2. If the type of the parameter is an open ARRAY OF T [6.2], the type of the argument may be any array with element type T1, where T and T1 are equal or the types T and T1 themselves observe this rule (with T as parameter type and T1 as argument type).

3. If the type of the parameter is ARRAY OF CHAR, the argument may be a *string*. The string is copied into an (anonymous) array variable which is passed as argument instead.

10.6 Array Constructors

An array constructor is a range declaration [9] followed by an expression. An array constructor consisting of an n-dimensional range declaration specifies an array [6.2] with at least n dimensions. The range identifiers correspond to the dimensions of the array in the order of their occurence in the range declaration. The array lengths in each dimension are the lengths of the corresponding ranges.

```
ArrayConstructor = "[" RangeDeclaration ":" Expression "]".
```

An array constructor of the form

$[r_1 = ..., r_2 = ..., ... r_n = ... : E(r_1, r_2, ... r_n)]$

denotes an array of type ARRAY $l_1, l_2, ... l_n$ OF T, where the l_k's are the lengths of the ranges associated with the range identifiers r_k ($1 \le k \le n$) and T is the type of the expression E. The array element denoted by the indices $i_1, i_2, ... i_n$ is the element $E(i_1, i_2, ... i_n)$, i.e. the value of the expression $E(i_1, i_2, ... i_n)$ which is obtained when all range identifiers r_k are substituted by i_k ($1 \le k \le n$). The number of dimensions of the constructed array is n plus the number of dimensions of the element type T.

If $E(r_1, r_2, ... r_n)$ is a designator denoting a *variable*, then the array constructor denotes an *array variable*. Array variables may be used as arguments for open array parameters [10.5]. The following restrictions apply to designators $E(r_1, r_2, ... r_n)$ of array variable constructors:

1. Only a restricted form of designators is allowed, namely designators observing the following syntactic production:

   ```
   ArrayDesignator  =  QualIdent {"[" ExpressionList "]" | "."}
   ```

2. An expression in the expression list of such a designator must either contain no range identifier at all, or it must be transformable into the form r∗x + y where r is a range identifier and x and y contain no range identifiers (i.e. the expression must be *linear* in r).
3. None of the index expressions containing a range identifier r must denote the same (constant) value for all values of r (i.e. x must not be zero in the expression r∗x + y; cf. rule 2).
4. All range identifiers introduced by the preceeding range declaration must be used at least once in the designator.

If $E(r_1, r_2, ... r_n)$ is an expression denoting a *value*, then the array constructor denotes an *array value*. Array values may be used as arguments for the predefined procedures SUM and PROD only.

Examples (see also examples in [7]):

Array constructor	*Denoted Array*
[i = 0 .. 3: id[2∗i + 1]]	elements id[1], id[3], id[5], id[7] (variable)

[j = 0 .. LEN(A)–1: A[j, j]]	diagonal of A (variable)
[k = 0 .. N–1: a[k] ∗ b[k]]	elementwise product of a and b (value)

10.7 Operators

Operators are special characters, character pairs or reserved words [3.5] denoting unary and binary operations. The operation denoted by an operator may depend on the type(s) of its operand(s). There are 4 different precedence classes for operators. The operator ~ has the highest precedence, followed by the multiplication operators, the addition operators and the relations. Operators of the same precedence associate from left to right; e.g. x–y–z is an abbreviation for (x–y)–z.

```
Expression = SimpleExpression [Relation SimpleExpression].
Relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
SimpleExpression = ["–"] Term {AddOperator Term}.
AddOperator = "+" | "–" | "%" | "|".
Term = ["/"] Factor {MulOperator Factor}.
MulOperator = "∗" | "/" | DIV | MOD | "&".
Factor =
  Literal | Set | Designator | FunctionCall | "~" Factor | "(" Expression ")".
```

10.7.1 Arithmetic Operators

The operators +, –, ∗ and / apply to operands of numeric types [6.1]. The result type is the smallest type which includes the types of both operands, except for division (/), where the result type includes the type REAL too. The expression 0–e may be abbreviated to –e.

Symbol	*Result*
+	sum
–	difference
∗	product
/	quotient
DIV	integer quotient
MOD	modulus

The operators DIV and MOD apply to integer operands only. For any dividend x and positive divisor y the following relationships hold:

$$x \equiv (x \text{ DIV } y) * y + x \text{ MOD } y$$
$$0 \leq x \text{ MOD } y < y$$

10.7.2 Logical Operators

Logical operators apply to boolean operands [6.1] and yield a boolean result. The evaluation of the operands of & and | proceeds from left to right and only as many operands are evaluated as necessary for the determination of the result.

Symbol	*Result*	*Description*
|	disjunction	p | q ≡ if p then TRUE else q
&	conjunction	p & q ≡ if p then q else FALSE
~	negation	~p ≡ not p

10.7.3 Set Operators

Set operators apply to operands of type SET [6.1] and yield a result of this type. A set expression of the form {0 .. MAX(SET)} / x may be abbreviated to /x. Note that x / y ≡ x * (/y).

Symbol	*Result*	*Description*
+	union	x IN s + t ≡ (x IN s) | (x IN t)
*	intersection	x IN s * t ≡ (x IN s) & (x IN t)
/	difference	x IN s / t ≡ (x IN s) & ~(x IN t)
%	symmetric difference	x IN s % t ≡ (x IN s) # (x IN t)

10.7.4 Relations

Relations yield a boolean result [6.1]. The ordering relations <, <=, > and >= apply to operands of type CHAR, INTEGER and REAL [6.1] as well as to character arrays [6.2] and strings [3.4]. The relations = and # apply to all unstructured types [6] (including the value NIL) as well as to character arrays and strings. x IN s tests whether x is an element of s. x must be of type INTEGER and s of type SET.

Symbol	*Relation*
=	equal
#	unequal
<	less
<=	less or equal
>	greater
>=	greater or equal
IN	element test
IS	type test

If the variable v is of record type [6.3], the *type test* v IS T holds if the dynamic type of v is an extension of T. If v is a pointer type [6.4], v IS T holds if the dynamic type of v↑ is an extension of T. The type test is applicable, if

1. v is a record variable parameter or v is a pointer (v ≠ NIL), and if
2. the record type T is an extension of the static type of v (or v↑ respectively).

Examples (see also examples in [7]):

Expression	*Type*
ch	CHAR
1993	INTEGER
i + j DIV 2	INTEGER
s / {0 .. 9, k}	SET
i + j / k	REAL
x * (–y) + k	REAL
f(i * x + 1.0E–3)	REAL
(u + v) * 1.0i	COMPLEX

2*(u + RE(v))	COMPLEX
~ok | (i # j)	BOOLEAN
('a' <= ch) & (ch <= 'z')	BOOLEAN
root.name < "lambda"	BOOLEAN
k IN {i .. j–1}	BOOLEAN
p.car IS Pair	BOOLEAN

11 Statements

A statement denotes an action. *Executing* a statement means performing the denoted action. There are *elementary* statements and *structured* statements. The former are not composed of any parts that are themselves statements. They are the assignment, the procedure call and the return statement. Structured statements are composed of parts that are themselves statements; they are used to express conditional, repetitive or parallel execution of its component statements.

Let P and Q be predicates denoting the program states before and after the execution of a statement S; i.e. P denotes a *precondition* of S and Q denotes a *postcondition*. Then the *predicate* {P} S {Q} holds, if P holds before the execution of S, S terminates and Q holds after the execution of S. An *axiom* of a structured statement S specifies rules for its component statements S_i and the predicates P and Q, such that {P} S {Q} holds.

```
Statement =
  Assignment | ProcedureCall | ReturnStatement |
  IfStatement | CaseStatement |
  WhileStatement | RepeatStatement | AllStatement.
```

11.1 Statement Sequences

A statement sequence denotes sequential execution of the component statements. Any two textually consecutive statements must be *separated* by a semicolon; the statement before the semicolon is executed before the statement following the semicolon. An empty statement sequence denotes no action at all.

```
StatementSequence = [Statement {";" Statement}].
```

Axiom of statement sequences:

$\{P\}\ S_1;\ S_2;\ \dots\ S_n\ \{Q\}$

holds if there exist conditions P_i and Q_i such that

$\forall\, i :\ 1 \le i \le n :\ \{P_i\}\ S_i\ \{Q_i\}$
$\forall\, i :\ 1 < i \le n :\ Q_{i-1} \Rightarrow P_i$
$P \Rightarrow P_1$
$Q_n \Rightarrow Q$

11.2 Assignments

The execution of an assignment replaces the current value of a variable [7] by the value of the specified expression [10].

```
Assignment = Designator ":=" Expression.
```

The type T(v) of the variable v in an assignment of the form v := e must be an unstructured type [6] or a character array [6.2], and one of the following conditions must hold for T(v) and the type of the expression T(e):

1. T(v) and T(e) are equal; or
2. T(v) and T(e) are numerical types and T(v) includes T(e); or
3. T(v) and T(e) are pointer types and the pointer base type of T(e) is an extension of the pointer base type of T(v), or
4. T(v) is a pointer or procedure type and e is NIL, or
5. T(v) is a character array and T(e) is a character array or e is a string. Then, e is copied to v and, if necessary, e is shortened to the length of v. The character sequence in v is always terminated by the character 0X.

Examples (see also examples in [7]):

```
ch := CR
i := 0
x := RE(u)
y := 1 + x*pi
k := Size(root)
```

```
    u := u / ABS(u)
    ok := (i < j) & (k IN {i .. j})
    A[i, j] := A[i, k] * v
    p := NIL
    q := q.cdr{Pair}
    root.left.name[0] := 0X
```

11.3 *Procedure Calls*

The execution of a procedure call activates the designated procedure. The call consists of a designator denoting a proper procedure [8] possibly followed by a list of arguments [10.5].

```
    ProcedureCall  =  Designator ["(" ArgumentList ")"].
```

Examples (see also examples in [7] and [8]):

```
    ReadInt(i)
    WriteInt(ORD(ch))
    saxpy(x, [i = 10 .. 20: a[i]], [i = 10 .. 20: b[i]])
    INC(root.mark)
    EXCL(s, 0)
```

11.4 *RETURN Statement*

A RETURN statement consists of the symbol RETURN, possibly followed by an expression. It indicates the termination of a procedure, and the expression specifies the result of a function procedure. Its type must be equal to the result type specified in the procedure heading. The expression and the result type must observe the rules for assignments.

```
    ReturnStatement  =  RETURN [Expression].
```

Function procedures require the presence of a RETURN statement indicating the result value. There may be several, although only one will be executed. In proper procedures, a RETURN statement is implied by the end of the procedure body. An explicit RETURN statement therefore appears as an additional (probably exceptional) termination point.

11.5 IF Statement

The IF statement specifies the conditional execution of *guarded* statements. The statement sequence following the symbol THEN is executed if the preceding boolean expression (guard) evaluates to true, otherwise the statement sequence following the symbol ELSE is executed, if there is one.

```
IfStatement  =
   IF Expression THEN StatementSequence
   {ELSIF Expression THEN StatementSequence}
   [ELSE StatementSequence]
   END.
```

An IF statement of the form

```
IF B1 THEN S1
ELSIF B2 THEN S2
...
ELSIF Bn THEN Sn
ELSE S0
END
```

is an abbreviation for

```
IF B1 THEN S1
ELSE
   IF B2 THEN S2
   ELSE
   ...
         ELSE Bn THEN Sn
         ELSE S0
         END
   ...
   END
END
```

Axiom of the IF statement:

$\{P\}$ IF B THEN S_1 ELSE S_2 END $\{Q\}$

holds if there exist conditions P_1 and P_2 such that

$\{P_1\}\ S_1\ \{Q\}$
$\{P_2\}\ S_2\ \{Q\}$
$P \wedge B \Rightarrow P_1$
$P \wedge \neg B \Rightarrow P_2$

Example:

```
IF ('0' <= ch) & (ch <= '9') THEN d := ORD(ch) – ORD('0')
ELSIF ('A' <= ch) & (ch <= 'F') THEN d := ORD(ch) – ORD('A')
ELSE Error; d := 0
END
```

11.6 CASE Statement

A CASE statement specifies the conditional execution of a statement sequence selected by the value of an expression. After evaluation of the expression, the statement sequence is executed, whose case label list contains the obtained value. The case expression and all case labels must be of the same type which must be CHAR or INTEGER. Case labels must be constant expressions and no value must occur more than once. If the value of the case expression does not occur as a label of any case, the statement sequence following the symbol ELSE is executed, if there is one. Otherwise the program is aborted.

```
CaseStatement =
   CASE Expression
   {OF CaseLabelList THEN StatementSequence}
   [ELSE StatementSequence]
   END.
CaseLabelList = CaseLabels {"," CaseLabels}.
CaseLabels = ConstExpression [".." ConstExpression].
```

Axiom of the CASE statement (each k_i stands for a single case label):

```
{P}
CASE k
OF $k_1$ THEN $S_1$
OF $k_2$ THEN $S_2$
...
OF $k_n$ THEN $S_n$
ELSE $S_0$
END
{Q}
```

holds, if there exist conditions P_i such that

$\forall i : 0 \le i \le n : \{P_i\}\ S_i\ \{Q\}$
$\forall i : 0 < i \le n : P \wedge (k = k_i) \Rightarrow P_i$
$\forall i : 0 < i \le n : P \wedge (k \ne k_i) \Rightarrow P_0$

Example:

```
CASE ch
OF '(' THEN sym := lparen
OF 27X THEN sym := quote
OF ')' THEN sym := rparen
OF '.' THEN sym := period
OF '0' .. '9' THEN Number
OF 'A' .. 'Z', 'a' .. 'z' THEN Identifier
ELSE sym := null
END
```

11.7 WHILE Statement

A WHILE statement specifies the repeated execution of a statement sequence as long as the preceding boolean expression (guard) evaluates to true. The statement sequence may not be executed at all.

```
WhileStatement = WHILE Expression DO StatementSequence END.
```

Axiom of the WHILE statement:

{P} WHILE B DO S END {Q}

holds if an invariant H exists such that

$P \Rightarrow H$
$\{H \wedge B\}$ S $\{H\}$
$H \wedge \neg B \Rightarrow Q$

Examples:

```
WHILE x > 0 DO
  IF ODD(x) THEN INC(i) END;
  x := x DIV 2
END

WHILE (q # NIL) & (q.name # name) DO p := q; q := q.next END
```

11.8 REPEAT Statement

The REPEAT statement specifies the repeated execution of a statement sequence until a condition is satisfied. The statement sequence is executed at least once.

```
RepeatStatement  =  REPEAT StatementSequence UNTIL Expression.
```

Axiom of the REPEAT statement:

{P} REPEAT S UNTIL B {Q}

holds if there exists an invariant H such that

$\{P\}$ S $\{H\}$
$\{H \wedge \neg B\}$ S $\{H\}$
$H \wedge B \Rightarrow Q$

Example:

```
REPEAT
  d[i] := ORD(x MOD 10 + ORD('0'));
  x := x DIV 10; INC(i)
UNTIL x = 0
```

11.9 ALL Statement

The ALL statement specifies the *potentially parallel* execution of a set of assignment sequences determined by an n-dimensional range declaration [9] which preceds the assignments. For each element of the set $R = \{(i_1, i_2, \ldots i_n) : (i_k \in ran(r_k)) \wedge (1 \leq k \leq n)\}$ where $ran(r_k)$ is the range associated with the range identifier r_k, the assignment sequence is executed exactly once. For a particular assignment sequence, the range identifier r_k stands for the value i_k of the element $(i_1, i_2, \ldots i_n)$ corresponding to that assignment sequence.

```
AllStatement = ALL RangeDeclaration DO AssignmentSequence END.
AssignmentSequence = [Assignment {";" Assignment}].
```

Let i denote $(i_1, i_2, \ldots i_n)$ for short. If S(i) denotes a particular assignment sequence (i.e. the assignment sequence corresponding to the element $(i_1, i_2, \ldots i_n) \in R$), then S(i) must be *independent* of any other assignment sequence S(j) with $i \neq j$; i.e. the order in which a range identifier r_k assumes a value within its associated range $ran(r_k)$ is *unspecified*. To be precise, let in(i) be the set of variables accessed by S(i) and out(i) be the set of variables to which values have been assigned to by S(i). Then, for any two different elements $i, j \in R$ the following must hold:

1. $in(i) \cap out(j)$ must be the empty set, and
2. $out(i) \cap out(j)$ must be the empty set.

Axiom of the ALL statement:

{P} ALL $i \in R$ DO S(i) END {Q}

holds if conditions P_i and Q_i exist such that

$P \Rightarrow (\forall\, i :\; i \in R :\; P_i)$
$\forall\, i :\; i \in R :\; \{P_i\}\; S(i)\; \{Q_i\}$
$(\forall\, i :\; i \in R :\; Q_i) \Rightarrow Q$

and

$\forall\, i, j :\; i, j \in R,\, i \neq j :\; (in(i) \cap out(j) = \emptyset) \wedge (out(i) \cap out(j) = \emptyset)$

with in(i) ≡ set of variables that have been accessed by S(i)
and out(i) ≡ set of variables to which values have been assigned to by S(i)

Examples (see also examples in [7]):

```
ALL i = 0 .. LEN(c)–1 DO c[i*2] := i * pi END
ALL i = j .. k DO a[j] := SELECT(a[j] > 0, a[j], 0) * b[j] END
ALL i, j = 0 .. LEN(A)–1 DO A[i, j] := 1/(i+j+1) END
ALL r = 10 .. 20 DO a[r] := a[index[r]]; a[index[r]] := 0 END
```

12 Modules

A *module* is a collection of declarations of objects and a sequence of statements for the purpose of assigning initial values to variables. A module constitutes a text that is compilable as a unit.

A module consists of a *module heading* and a *body*. The heading defines the module identifier and possibly *imported* modules. The body contains declarations and statements that are executed, when the module is added to a system (loaded). The module identifier must be repeated at the end of the module specification.

```
Module = ModuleHeading ";" Body END ident ".".
ModuleHeading = MODULE ident [ImportList].
ImportList = "(" Import {"," Import} ")".
Import = Ident [":=" ident].
```

The import list specifies the modules which are *imported* and hence of which the importing module is a *client*. If an identifier x is *exported* [4] by a module M, and M appears in a module's import list, then x is referred to as M.x within the importing module. An import of the form M1 := M may be used to (locally) substitute the module identifier M by M1. An identifier x exported by the module *M* is then referred to as *M1.x* within the importing module.

The values of global variables are undefined upon loading of the module. The statements in bodies of imported modules are executed before any statement of the importing module is executed.

Example:

```
MODULE Out (Unix);

  CONST maxLen = 80;
  VAR len: INTEGER;

  PROCEDURE Ln*;
  BEGIN Unix.Put(0AX); Unix.Flush; len := 0
  END Ln;

  PROCEDURE Char* (x: CHAR);
  BEGIN
    IF len = maxLen THEN Ln END;
    Unix.Put(ch); INC(len)
  END Char;

  PROCEDURE String* (VAR s: ARRAY OF CHAR);
    VAR i: INTEGER;
  BEGIN i := 0;
    WHILE (i < LEN(s)) & (s[i] # 0X) DO Char(s[i]); INC(i) END
  END String;

  PROCEDURE Int* (x: INTEGER);
    VAR i: INTEGER; d: ARRAY 16 OF CHAR;
  BEGIN i := 0;
    IF x < 0 THEN x := –x; Char('–') END;
    REPEAT d[i] := CHR(x MOD 10 + 30H); x := x DIV 10; INC(i)
    UNTIL x = 0;
    REPEAT DEC(i); Char(d[i]) UNTIL i = 0
  END Int;
```

```
    PROCEDURE Set* (s: SET);
      VAR i, j: INTEGER;
    BEGIN
      Char('{'); j := 0;
      WHILE s # {} DO
        IF j IN s THEN i := j;
          REPEAT EXCL(s, j); INC(j) UNTIL (s = {}) | ~(j IN s);
          Int(i);
          IF i+2 = j THEN String(", "); Int(j–1)
          ELSIF i+2 < j THEN String(" .. "); Int(j–1)
          END;
          IF s # {} THEN String(", ") END
        END;
        INC(j)
      END;
      Char('}')
    END Set;

  BEGIN Unix.Init; len := 0
  END Out.
```

12.1 The Module SYSTEM

The module SYSTEM provides procedures that are necessary to program certain *low level* operations and to circumvent the type system of Oberon-V. A module importing SYSTEM must be considered to be inherently *unsafe* and *non-portable*; therefore SYSTEM should be used very restrictively. Although the operations provided by it are machine specific and hence may vary from machine to machine, the following procedures are assumed to be found in every SYSTEM implementation. x, y and n stand for expressions, v stands for a variable and T for a type.

Function procedures:

Call	*Argument types*	*Result type*	*Function*
ADR(v)	any type	INTEGER	address of variable v
LSL(x, n)	INTEGER	INTEGER	logical shift left by n bits
LSR(x, n)	INTEGER	INTEGER	logical shift right by n bits

SIZE(T)	any type	INTEGER	space required by T
VAL(T, x)	any type	type of T	x interpreted as of type T

Proper procedures:

Call	*Argument types*	*Function*
GET(a, v)	a: INTEGER; v: any unstructured type	v := Mem[a]
PUT(a, x)	a: INTEGER; v: any unstructured type	Mem[a] := x
MOVE (x, y, n)	INTEGER	ALL i = 0 .. n–1 DO Mem[y+i] := Mem[x+i] END
NEW(v, n)	v: any pointer type n: integer type	allocate storage block of n words and assign its address to v

Appendix B: Program Examples

The following program examples shall demonstrate the usage of ALL statements and of array constructors. The BLA routines as well as the Linpack procedures have been adapted from [Dongarra 1979].

1 BLAS

Module OVBlas contains five of the most frequently used BLA subroutines. This module has been used for the measurements in Section 6.1.

```
MODULE OVBlas;

  TYPE
    RealVector* = ARRAY OF REAL;

  PROCEDURE isamax* (VAR x: RealVector): INTEGER;
    VAR i, j: INTEGER; max: REAL;
  BEGIN max := ABS(x[0]); i := 0; j := 1;
    WHILE j < LEN(x) DO
      IF ABS(x[j]) > max THEN max := ABS(x[j]); i := j END;
      INC(j)
    END;
    RETURN i
  END isamax;

  PROCEDURE sscal* (a: REAL; VAR x: RealVector);
  BEGIN ALL i = 0 .. LEN(x)-1 DO x[i] := a * x[i] END
  END sscal;
```

```
    PROCEDURE saxpy* (a: REAL; VAR x, y: RealVector);
    BEGIN
       ASSERT(LEN(x) = LEN(y));
       ALL i = 0 .. LEN(x)–1 DO y[i] := a * x[i] + y[i] END
    END saxpy;

    PROCEDURE sasum* (VAR x: RealVector): REAL;
    BEGIN RETURN SUM([i = 0 .. LEN(x) – 1: ABS(x[i])])
    END sasum;

    PROCEDURE sdot* (VAR x, y: RealVector): REAL;
    BEGIN RETURN SUM([i = 0 .. LEN(x) –1: x[i] * y[i]])
    END sdot;

 END OVBlas.
```

2 Linpack

Module OVLinpack1 contains a one-to-one translation of the Linpack procedures *sgefa* and *sgesl*. They are used to solve general systems of simultaneous linear algebraic equations. For a detailed discussion of the algorithms the reader is referred to [Dongarra 1979].

```
 MODULE OVLinpack1 (Blas := OVBlas);

    TYPE
       IntVector* = ARRAY OF INTEGER;
       RealVector* = Blas.RealVector;
       RealMatrix* = ARRAY OF RealVector;

    PROCEDURE sgefa* (VAR a: RealMatrix; VAR ipvt: IntVector; VAR info: INTEGER);
       VAR
          j, k, kp1, l, n, nm1: INTEGER;
          t: REAL;
    BEGIN
       info := –1; n := LEN(a); nm1 := n–1;
       IF nm1 >= 1 THEN k := 0;
          WHILE k < nm1 DO kp1 := k+1;
             (* find l = pivot index *)
             l := Blas.isamax([i = k .. nm1: a[i, k]]) + k;
             ipvt[k] := l;
             (* zero pivot implies this column already triangularized *)
             IF a[l, k] # 0 THEN
                (* interchange if necessary *)
                IF l # k THEN t := a[l, k]; a[l, k] := a[k, k]; a[k, k] := t END;
                (* compute multipliers *)
                t := -1/a[k, k];
                Blas.sscal(t, [i = kp1 .. nm1: a[i, k]]);
                (* row elimination with column indexing *)
                j := kp1;
                WHILE j < n DO t := a[l, j];
                    IF l # k THEN a[l, j] := a[k, j]; a[k, j] := t END;
                    Blas.saxpy(t, [i = kp1 .. nm1: a[i, k]], [i = kp1 .. nm1: a[i, j]]);
                    INC(j)
                END
            ELSE info := kp1
            END;
            k := kp1
        END
    END;
    ipvt[nm1] := nm1;
    IF a[nm1, nm1] = 0 THEN info := n END
END sgefa;

PROCEDURE sgesl*
    (VAR a: RealMatrix; VAR ipvt: IntVector; VAR b: RealVector; job: INTEGER);
    VAR
        n, k, kb, l, nm1: INTEGER;
        t: REAL;
BEGIN
    n := LEN(a); nm1 := n-1;
    IF job = 0 THEN
        (* solve a * x = b; first solve l * y = b *)
        IF nm1 >= 1 THEN k := 0;
            WHILE k < nm1 DO
                l := ipvt[k]; t := b[l];
                IF l # k THEN b[l] := b[k]; b[k] := t END;
                Blas.saxpy(t, [i = 1 .. n-k-1: a[k+i, k]], [i = 1 .. n-k-1: b[k+i]]);
                INC(k)
            END
        END;
        (* now solve u * x = y *)
        kb := 0;
        WHILE kb < n DO
            k := n - 1 - kb;
            b[k] := b[k] / a[k, k];
            t := -b[k];
            Blas.saxpy(t, [i = 0 .. k-1: a[i, k]], [i = 0 .. k-1: b[i]]);
            INC(kb)
        END
    ELSE
        (* solve trans(u) * x = b; first solve trans(u) * y = b *)
        k := 0;
```

```
            WHILE k < n DO
               t := Blas.sdot([i = 0 .. k–1: a[i, k]], [i = 0 .. k–1: b[i]]);
               b[k] := (b[k] – t) / a[k, k];
               INC(k)
            END;
            (* now solve trans(l) * x = y *)
            IF nm1 >= 1 THEN kb := 0;
               WHILE kb < nm1 DO
                  k := n – 1 – kb;
                  b[k] := Blas.sdot([i = 1 .. n–k–1: a[k+i, k]], [i = 1 .. n–k–1: b[k+i]]) + b[k];
                  l := ipvt[k];
                  IF l # k THEN t := b[l]; b[l] := b[k]; b[k] := t END;
                  INC(kb)
               END
            END
         END
      END sgesl;

   END OVLinpack1.
```

Module OVLinpack2 contains the same procedures as OVLinpack1, but with "inlined" ALL statements instead of the BLAS calls. Because the BLA procedures reduce to simple "one-liners" when using ALL statements, this leads to a better readable program (only the procedure sgefa is shown). This module has been used for the measurements in Section 6.2.

```
   MODULE OVLinpack2 (Blas := OVBlas);

      TYPE
         IntVector* = ARRAY OF INTEGER;
         RealVector* = Blas.RealVector;
         RealMatrix* = ARRAY OF RealVector;

      PROCEDURE sgefa* (VAR a: RealMatrix; VAR ipvt: IntVector; VAR info: INTEGER);
         VAR j, k, l, n: INTEGER; t: REAL;
      BEGIN
         info := –1; n := LEN(a);
         IF n > 1 THEN k := 0;
            WHILE k < n–1 DO;
               (* find l = pivot index *)
               l := Blas.isamax([i = k .. n–1: a[i, k]]) + k ;
               ipvt[k] := l;
               (* zero pivot implies this column already triangularized *)
               IF a[l, k] # 0 THEN
                  (* interchange if necessary *)
                  IF l # k THEN t := a[l, k]; a[l, k] := a[k, k]; a[k, k] := t; END;
```

```
                (* compute multipliers *)
                t := -1 /a[k, k];
                ALL i = k+1 .. n-1 DO a[i, k] := t*a[i, k] END;
                (* row elimination with column indexing *)
                j := k+1;
                WHILE j < n DO t := a[l, j];
                  IF l # k THEN a[l, j] := a[k, j]; a[k, j] := t END;
                  ALL i = k+1 .. n-1 DO a[i, j] := a[i, j] + t*a[i, k] END;
                  INC(j)
                END
              ELSE info := k
              END;
              INC(k)
            END
          END;
          ipvt[n-1] := n-1;
          IF a[n-1, n-1] = 0 THEN info := n-1 END
        END sgefa;

      END OVLinpack2.
```

3 Factorial

Module Factorial computes all digits of n factorial (n!). The result is stored in an array X of length L. X contains digits to the base B (1000000), i.e. the value of X is:

$$X[L-1]*B^{L-1} + X[L-2]*B^{L-2} + ... + X[1]*B^1 + X[0]*B^0$$

The result is obtained by multiplying the value of X with n, n–1, ... 2, starting with X[0] = 1 and L = 1 (procedure BigFact). A special multiply step (procedure MultiplyStep) is used, which computes result digits and carrys separately. This subdivision allows for using the ALL statement. In order to obtain better performance on the Cray Y-MP, real arithmetic is used instead of integer arithmetic. The operations DIV and MOD are "emulated" using the following relationships:

$x \text{ DIV } y = \text{TRUNC}((x + 0.5) / y)$ (for $x \geq 0$ and $y > 0$)
$x \text{ MOD } y = x - y * \text{TRUNC}((x + 0.5) / y)$ (for $x \geq 0$ and $y > 0$)

Since machine division (/) is not exact, a correction value is required (0.5). The result is correct, if $\varepsilon < 1 / 2y$, where ε is the absolute error of the machine division x/y (without proof). The same method is used by the Oberon-V compiler to implement the integer operations DIV and MOD.

```
MODULE Factorial (Out, Timer);

  CONST B = 1000000; logB = 6;

  VAR
    X, C1, C2, H: ARRAY 100000 OF REAL;
    L: INTEGER;

  PROCEDURE MultiplyStep (Y: INTEGER);
  BEGIN
  (* independant digit multiply *)
    C2[0] := 0;
    ALL i = 0 .. L–1 DO
      H[i] := (X[i] + C1[i]) * Y;
      C2[i+1] := TRUNC((H[i] + 0.5) / B); (* real DIV *)
      X[i] := H[i] – B * C2[i+1] (* real MOD *)
    END;
    IF C2[L] > 0 THEN X[L] := 0; L := L+1 END;
  (* independant carry reduction *)
    C1[0] := 0;
    ALL i = 0 .. L–1 DO
      H[i] := X[i] + C2[i];
      C1[i+1] := TRUNC((H[i] + 0.5) / B); (* real DIV *)
      X[i] := H[i] – B * C1[i+1] (* real MOD *)
    END;
    IF C1[L] > 0 THEN X[L] := 0; L := L+1 END
  END MultiplyStep;

  PROCEDURE CleanUp;
    VAR i: INTEGER; h, c: REAL;
  BEGIN i := 0; c := 0;
    WHILE i < L DO
      h := X[i] + C1[i] + c;
      c := TRUNC((h + 0.5) / B); (* real DIV *)
      X[i] := h – B*c; (* real MOD *)
      i := i+1
    END;
    IF c > 0 THEN X[L] := c; L := L+1 END
  END CleanUp;

  PROCEDURE BigFact (n: INTEGER);
  BEGIN L := 1; X[0] := 1; C1[0] := 0;
```

```
      WHILE n > 1 DO MultiplyStep(n); n := n–1 END;
      CleanUp
   END BigFact;

   PROCEDURE WriteInt (x: INTEGER; n: INTEGER); (* x >= 0 *)
      VAR i: INTEGER; d: ARRAY 10 OF CHAR;
   BEGIN i := 0;
      REPEAT d[i] := CHR(x MOD 10 + 30H); x := x DIV 10; i := i+1 UNTIL x = 0;
      WHILE n > i DO n := n–1; Out.Char('0') END;
      WHILE i > 0 DO i := i–1; Out.Char(d[i]) END
   END WriteInt;

   PROCEDURE WriteRes;
      VAR i, i0, count: INTEGER;
   BEGIN
      IF L > 0 THEN i := L–1; WriteInt(TRUNC(X[i]), logB); count := logB; i0 := i–100;
         IF i0 < 0 THEN i0 := 0 END;
         WHILE i > i0 DO i := i–1; WriteInt(TRUNC(X[i]), logB); count := count + logB;
            IF count >= 72 THEN Out.Ln; count := 0 END
         END;
         IF i0 > 0 THEN Out.String("...") END;
         Out.String(" (L = "); Out.Int(L); Out.Char(')')
      END;
      Out.Ln
   END WriteRes;

   PROCEDURE Fact (n: INTEGER);
   BEGIN ASSERT(n < B);
      WriteInt(n, 0); Out.String("! = "); BigFact(n); WriteRes; Out.Ln
   END Fact;

BEGIN
   Timer.Reset;
   Timer.Start; Fact(1000); Timer.Stop;
   Timer.Times; Out.Ln
END Factorial.
```

All digits of 1000! can be computed in 28ms on the Cray Y-MP:

```
1000! = 4023872600770937735437024339230039857193748642107146325437999 1
0429938512398629020592044208486969404800479988610197196058631666872 9
9480855890132382966994459099742450408707375991882362772718873251977 9
5059509952761208749754624970436014182780946464962910563938874378864 8
7337119181045825783647849977012476632889835955735432513185323958463 0
75557409114262417474349347553428646576611667797396668820291... (L = 428)

time = 0.0279850799440299s  clocks = 4664180
```