

9. A RISC-Architecture as Target

It is worth noting that our compiler, up to this point, could be developed without reference to the target computer for which it is to generate code. But why indeed should the target machine's structure influence syntactic analysis and error handling? On the contrary, such an influence should consciously be avoided. As a result, code generation for an arbitrary computer may be added according to the principle of stepwise refinement to the existing, machine independent parser, which serves like a scaffolding. Before undertaking this task, however, a specific target architecture must be selected.

To keep both the resulting compiler reasonably simple and the development clear of details that are of relevance only for a specific machine and its idiosyncrasies, we postulate an architecture according to our own choice. Thereby we gain the considerable advantage that it can be tailored to the needs of the source language. This architecture does exist as a real machine, implemented on a field-programmable gate array (FPGA), described in full detail as a text in the hardware design language Verilog. But it is also described by a program called an *emulator*. A real computer may then be used to execute this program, forming a *virtual machine* which interprets the generated code.

It is not the aim of this text to present the motivations for our choice of the specific virtual architecture with all its details. This chapter is rather intended to serve as a descriptive manual consisting of an informal introduction and a semi-formal definition of the computer in the form of the interpretive program. The emulator may be used in cases where the actual computer is not available.

In the definition of this computer we intentionally follow closely the line of RISC-architectures. The acronym RISC stands for *reduced instruction set computer*, where "reduced" is to be understood as relative to architectures with large sets of complex instructions, as these were dominant until about 1980. This is obviously not the place to explain the essence of the RISC architecture, nor to expound on its various advantages. Here it is attractive because of its simplicity and clarity of concepts, which simplify the description of the instruction set and the choice of instruction sequences corresponding to specific language constructs. The architecture chosen here is similar to the one presented by Hennessy and Patterson (1990) under the name DLX. The small deviations are due to our desire for increased regularity. Among commercial products, the MIPS and ARM architectures are closest to ours.

9.1. Resources and registers

An architecture defines those aspects of a computer that are relevant to the programmer and the compiler designer. A computer consists of an arithmetic unit, a control unit and a store. Our arithmetic unit contains 16 registers R0 – R15, with 32 bits each. The control unit consists of the *instruction register* (IR), holding the instruction currently being executed, and the *program counter* (PC), holding the address of the instruction to be fetched next (Figure 9.1). Branch instructions to procedures implicitly use register R15 to store the return address. The memory consists of 32-bit words, and it is byte-addressed, that is, word addresses are multiples of 4. Furthermore, there are 4 single-bit status registers N, Z, C, and V called the *condition codes*.

There are three types of instructions and instruction formats. *Register instructions* operate on registers only and feed data through the arithmetic-logic unit ALU or through a shifter. *Memory instructions* move data between registers and memory. *Branch instructions* affect the program counter.

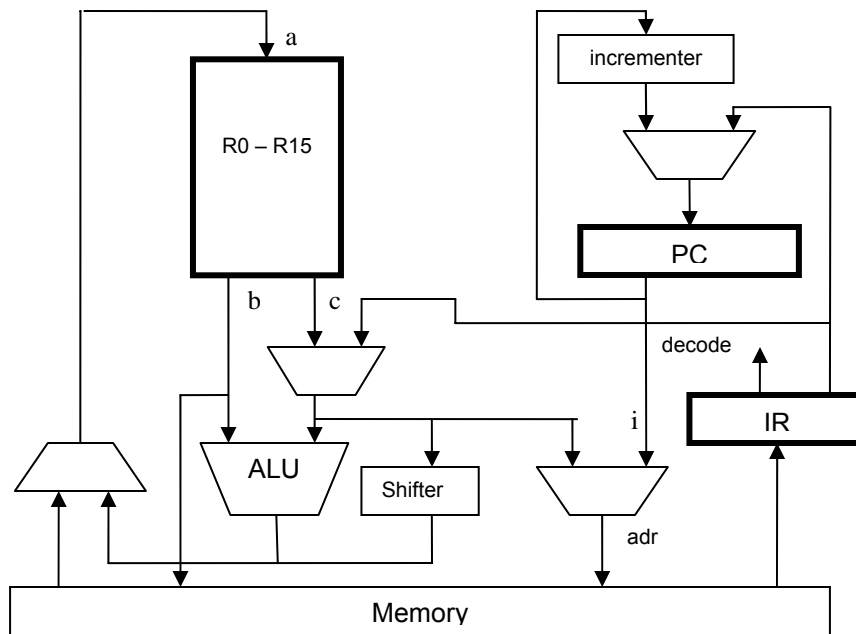


Figure 9.1. Block diagram of the RISC structure

9.2. Register instructions

These appear in two formats. In format *F0* the operands are *R.b* and *n (= R.c)*. In format *F1* the second operand *n* is not a register, but the constant *im*. In both formats, the result is assigned to register *R.a*.

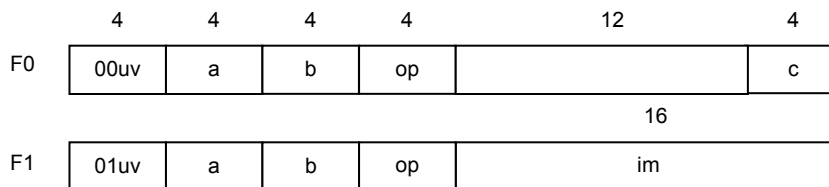


Fig.9.2. Formats F0 and F1. for register instructions

The operations provided are

0	MOV	a, n	$R.a := n$	
1	LSL	a, b, n	$R.a := R.b \leftarrow n$	(shift left by n bits)
2	ASR	a, b, n	$R.a := R.b \rightarrow n$	(shift right by n bits with sign extension)
3	ROR	a, b, n	$R.a := R.b \text{ rot } n$	(rotate right by n bits)
4	AND	a, b, n	$R.a := R.b \& n$	logical operations
5	ANN	a, b, n	$R.a := R.b \& \sim n$	
6	IOR	a, b, n	$R.a := R.b \text{ or } n$	
7	XOR	a, b, n	$R.a := R.b \text{ xor } n$	
8	ADD	a, b, n	$R.a := R.b + n$	integer arithmetic
9	SUB	a, b, n	$R.a := R.b - n$	
10	MUL	a, b, n	$R.a := R.a \times n$	
11	DIV	a, b, n	$R.a := R.b \text{ div } n$	

The field *im* is only 16 bits wide. It is extended to a 32-bit word according to the *v* modifier bit:

v = 0 extend with 16 zeroes

v = 1 extend with 16 ones

The four single-bit condition registers are affected as follows. These registers are tested by branch instructions.

- N := (R.a < 0) (Negative)
- Z := (R.a = 0) (Zero)
- C := carry out (for addition, subtraction, and comparison)
- V := overflow (for signed addition, subtraction, and comparison)

9.3. Memory instructions (format F2)

There are only two instructions accessing memory, load and store. It is a characteristic of the RISC structure that access to memory is not combined with any other operation. All arithmetic or logical operations are performed on registers.

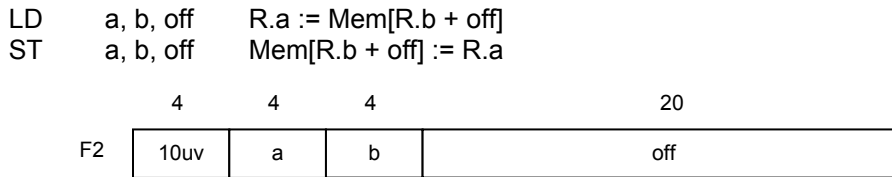


Fig. 9.3. Format F2 for memory instructions

The modifier bits have the following significance:

- u = 0 load, u = 1 store
- v = 0: word, v = 1: byte

9.4. Branch instructions (Format F3)

Branch instructions are used to break the sequence of instructions. The next instruction is designated either by a 24-bit (signed) offset, or by the value of a register, depending on the modifier bit *u*. It indicates the length of the jump forward or backward (PC-relative addressing). This offset is in *words*, not bytes, as instructions are always one word long.

- | | | | |
|-------|-----------|-------|----------------|
| Bcond | off | | |
| u = 0 | PC := R.c | u = 1 | PC := PC+1+off |
| v = 0 | no link | v = 1 | R15 := PC+1 |

The modifier *v* determines, whether the current value of PC be stored in register R15 (the *link* register). This facility is used for calls to procedures. The value stored is then the return address. The format is shown in Fig. 4.

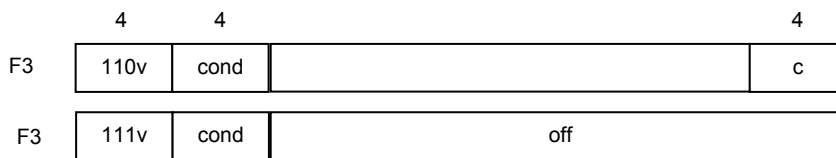


Fig. 9.4. Format F3 of branch instructions

The field *cond* determines, under which conditions the branch is executed. If not, the instruction has no effect. The selected condition is a logical function of the registers N, Z, C, and V. The following are available:

code	cond	condition	code	cond	condition
0000	MI	negative (minus)	N	1000	PL positive (plus) ~N
0001	EQ	equal (zero)	Z	1001	NE positive (plus) ~Z

0010	CS	carry set	C	1010	CC	carry clear	$\sim C$
0011	VS	overflow set	V	1011	VC	overflow clear	$\sim V$
0100	LS	less or same	$\sim C Z$	1100	HI	high	$\sim(\sim C Z)$
0101	LT	less than	$N \neq V$	1101	GE	greater or equal	$\sim(N \neq V)$
0110	LE	less or equal	$(N \neq V) Z$	1110	GT	greater than	$\sim((N \neq V) Z)$
0111		always	T	1111		never	F

9.5. An Emulator

An emulator of the RISC architecture is a program that simulates it, that interprets RISC instructions. The program listed below contains procedure *Execute*. It is called in the Oberon context by the command *OSG.Execute*. This command provides the RISC's memory as an array *M* into which the compiler deposited the compiled instructions. A few comments are in order:

1. The emulator describes only most of the RISC, and a few facilities are omitted.
2. The condition registers C and V are not considered. C is needed for unsigned arithmetic only. Conditions CS, CC, VS, VC, LS, and HI are ignored (see table above).
3. Byte access in memory instructions is not considered. (The v-bit is ignored).
4. Negative addresses are reserved for access to input and output devices. This common technique is called *memory mapping*. We follow this custom and emulate an input and an output channel represented by a scanner *S* and a writer *W*. *S* reads from the text following the command *RISC.Execute*, and *W* writes into the Oberon system's log text. The respective statements are:

```

ReadInt(x)      Read the next number and assign it to variable x,
eot()           The end of the input text had been reached. A Boolean function.
WriteInt(x)     Append integer x to the log text,
WriteChar(x)    Append the character with ordinal number x to the log text
WriteLn        Terminate the current line

```

The following simple program reads a sequence of integers and copies it to the output.

```

MODULE RISC; (*NW 22.9.07 / 28.10.2013*)
IMPORT SYSTEM, Texts, Oberon;
CONST
  MOV = 0; LSL = 1; ASR = 2; ROR = 3; AND = 4; ANN = 5; IOR = 6; XOR = 7;
  ADD = 8; SUB = 9; MUL = 10; Div = 11;

VAR IR: LONGINT; (*instruction register*)
    PC: LONGINT; (*program counter*)
    N, Z: BOOLEAN; (*condition flags*)
    R: ARRAY 16 OF LONGINT;
    H: LONGINT;

PROCEDURE Execute*(VAR M: ARRAY OF LONGINT; pc: LONGINT;
  VAR S: Texts.Scanner; VAR W: Texts.Writer);
  VAR a, b, op, im: LONGINT; (*instruction fields*)
      adr, A, B, C, n: LONGINT;
      MemSize: LONGINT;
BEGIN PC := 0; R[13] := pc * 4; R[14] := LEN(M)*4; n := 0;
  REPEAT (*interpretation cycle*)
    IR := M[PC]; INC(PC); INC(n);
    a := IR DIV 1000000H MOD 10H;
    b := IR DIV 100000H MOD 10H;
    op := IR DIV 10000H MOD 10H;
    im := IR MOD 10000H;
    IF  $\sim$ ODD(IR DIV 80000000H) THEN (*~p: register instruction*)
      B := R[b];
      IF  $\sim$ ODD(IR DIV 40000000H) THEN (*~q*) C := R[IR MOD 10H]
      ELSEIF  $\sim$ ODD(IR DIV 10000000H) THEN (*q&~v*) C := im

```

```

ELSE (*q&v*) C := im + 0FFFF0000H
END ;
CASE op OF
  MOV: IF ~ODD(IR DIV 20000000H) THEN A := C ELSE A := H END |
  LSL: A := SYSTEM.LSH(B, C) |
  ASR: A := ASH(B, -C) |
  ROR: A := SYSTEM.ROT(B, -C) |
  AND: A := SYSTEM.VAL(LONGINT, SYSTEM.VAL(SET, B) * SYSTEM.VAL(SET, C)) |
  ANN: A := SYSTEM.VAL(LONGINT, SYSTEM.VAL(SET, B) - SYSTEM.VAL(SET, C)) |
  IOR: A := SYSTEM.VAL(LONGINT, SYSTEM.VAL(SET, B) + SYSTEM.VAL(SET, C)) |
  XOR: A := SYSTEM.VAL(LONGINT, SYSTEM.VAL(SET, B) / SYSTEM.VAL(SET, C)) |
  ADD: A := B + C |
  SUB: A := B - C |
  MUL: A := B * C |
  Div: A := B DIV C; H := B MOD C
END ;
R[a] := A; N := A < 0; Z := A = 0
ELSIF ~ODD(IR DIV 40000000H) THEN (*p & ~q: memory instruction*)
  adr := (R[b] + IR MOD 100000H) DIV 4;
  IF ~ODD(IR DIV 20000000H) THEN
    IF adr >= 0 THEN (*load*) R[a] := M[adr]; N := A < 0; Z := A = 0
    ELSE (*input*)
      IF adr = -1 THEN (*ReadInt*) Texts.Scan(S); R[a] := S.i;
      ELSIF adr = -2 THEN (*eot*) Z := S.class # Texts.Int
      END
    END
  ELSE
    IF adr >= 0 THEN (*store*) M[adr] := R[a];
    ELSE (*output*)
      IF adr = -1 THEN Texts.WriteInt(W, R[a], 4)
      ELSIF adr = -2 THEN Texts.Write(W, CHR(R[a]))
      ELSIF adr = -3 THEN Texts.WriteLine(W); Texts.Append(Oberon.Log, W.buf)
      END
    END
  END
END
ELSE (* p & q: branch instruction*)
  IF (a = 0) & N OR (a = 1) & Z OR (a = 5) & N OR (a = 6) & (N OR Z) OR (a = 7) OR
    (a = 8) & ~N OR (a = 9) & ~Z OR (a = 13) & ~N OR (a = 14) & ~(N OR Z) THEN
    IF ODD(IR DIV 10000000H) THEN R[15] := PC * 4 END ;
    IF ODD(IR DIV 20000000H) THEN PC := (PC + (IR MOD 1000000H)) MOD 40000H
    ELSE PC := R[IR MOD 10H] DIV 4
    END
  END
END
UNTIL (PC = 0) OR (n = 8192);
Texts.Append(Oberon.Log, W.buf)
END Execute;
END RISC.

```

This design has been implemented on a single field-programmable gate array (FPGA) and is available on a low-cost Xilinx Spartan-3 development board.

10. Expressions and Assignments

10.1. Straight code generation according to the stack principle

The third example in Chapter 5 showed how to convert an expression from conventional infix form into its equivalent postfix form. Our ideal computer would be capable of directly interpreting postfix notation. As also shown, such an ideal computer requires a stack for holding intermediate results. Such a computer architecture is called a *stack architecture*.

Computers based on a stack architecture are not in common use. Sets of explicitly addressable registers are preferred to a stack. Of course, a set of registers can easily be used to emulate a stack. Its top element is indicated by a global variable representing the register stack index RH in the compiler. This is feasible, since the number of intermediate results is known at compile time, and the use of a global variable is justified because the stack constitutes a global resource.

To derive the program for generating the code corresponding to specific constructs, we first postulate the desired code patterns. This method will also be successfully employed later for other constructs beyond expressions and assignments. Let the code for a given construct *K* be given by the following table:

<u>K</u>	<u>code(K)</u>	<u>side effect</u>
ident	LDW RH, SB, adr(ident)	INC(RH)
number	MOV RH, value	INC(RH)
(exp)	code(exp)	
fac0 * fac1	code(fac0) code(fac1) MUL RH, RH, RH+1	DEC(RH)
term0 + term1	code(term0) code(term1) ADD RH, RH, RH+1	DEC(RH)
ident := exp	code(exp) STW RH, adr(ident)	DEC(RH)

To begin, we restrict our attention to simple variables as operands, and we omit selectors for structured variables. We assume global variables, whose base address shall be present in a reserved register SB (static base). First, consider the assignment $u := x*y + z*w$:

Instruction	encoding	meaning	stack	stack index RH
LDW R0, SB, x	80D00004	R0 := x	x	1
LDW R1, SB, y	81D00008	R1 := y	x, y	2
MUL R0, R0, R1	000A0001	R0 := R0*R1	x*y	1
LDW R1, SB, z	81D0000C	R1 := z	x*y, z	2
LDW R2, SB, w	82D00010	R2 := w	x*y, z, w	3
MUL R1, R1, R2	011A0002	R1 := R1 * R2	x*y, z*w	2
ADD R0, R0, R1	00080001	R0 := R0 + R1	x*y + z*w	1
STW R0, SB, u	A0D00000	u := R0	-	0

From this it is quite evident how the corresponding parser procedures must be extended. The following identifiers are used to denote the respective operation codes:

```
Mov = 0; Lsl = 1; Asr = 2; Ror = 3; And = 4; Ann = 5; lor = 6; Xor = 7;
Add = 8; Sub = 9; Cmp = 9; Mul = 10; Div = 11;
Ldw = 0; Stw = 2;

PROCEDURE factor;
  VAR obj: Object;
```

```

BEGIN
  IF sym = ident THEN find(obj); Get(sym); Put2(Ldw, RH, SB, obj.val); INC(RH)
  ELSIF sym = number THEN Put1(Mov, RH, 0, val); Get(sym); INC(RH)
  ELSIF sym = lparen THEN
    Get(sym); expression;
    IF sym = rparen THEN Get(sym) ELSE Mark(" ) missing") END
  ELSIF ...
  END
END factor;

PROCEDURE term;
  VAR op: INTEGER;
BEGIN factor;
  WHILE (sym = times) OR (sym = div) DO
    op := sym; Get(sym); factor;
    IF op = times THEN DEC(RH); Put0(Mul, RH-1, RH-1, RH)
    ELSIF op = div THEN DEC(RH); Put0(Div, RH-1, RH-1, RH)
    END
  END
END term;

PROCEDURE SimpleExpression;
  VAR op: INTEGER;
BEGIN
  IF sym = plus THEN Get(sym); term
  ELSIF sym = minus THEN
    Get(sym); term; Put1(Mov, RH+1, 0, 0); Put0(Sub, RH, RH+1, RH)
  ELSE term
  END ;
  WHILE (sym = plus) OR (sym = minus) DO
    op := sym; Get(sym); term;
    IF op = plus THEN DEC(RH); Put0(Add, RH-1, RH-1, RH)
    ELSIF op = minus THEN DEC(RH); Put0(Sub, RH-1, RH-1, RH)
    END
  END
END SimpleExpression;

PROCEDURE Statement;
  VAR obj: Object;
BEGIN
  IF sym = ident THEN
    find(obj); Get(sym);
    IF sym = becomes THEN
      Get(sym); expression; DEC(RH); Put2(Stw, RH, SB, obj.val)
    ELSIF ...
    END
  ELSIF ...
  END
END Statement;

```

Here we have introduced the generator procedure *Put*. In fact we use 4 such procedures, one for each class of instructions, They can be regarded as the counterpart of the scanner procedure *Get*. We assume that they deposit an instruction in a global array, using the variable *pc* as index denoting the next free location.

```

PROCEDURE Put0(op, a, b, c: LONGINT);
BEGIN (*emit register-register instruction*)
  code[pc] := ((a*10H + b) * 10H + op) * 10000H + c; INC(pc)
END Put0;

PROCEDURE Put1(op, a, b, im: LONGINT);
BEGIN (*emit register-immediate instruction*)
  IF im < 0 THEN INC(op, 1000H) (*set v bit*) END ;

```

```

code[pc] := (((a+40H) * 10H + b) * 10H + op) * 10000H + (im MOD 10000H); INC(pc)
END Put1;

PROCEDURE Put2(op, a, b, off: LONGINT);
BEGIN (*emit load/store instruction*)
code[pc] := (((op+8) * 10H + a) * 10H + b) * 100000H + (off MOD 10000H); INC(pc)
END Put2;

PROCEDURE Put3(op, cond, off: LONGINT);
BEGIN (*emit branch instruction*)
code[pc] := ((op+12) * 10H + cond) * 1000000H + (off MOD 1000000H); INC(pc)
END Put3;

```

Addresses of variables are indicated in the generated code listed above by simply using their identifier. In reality, the address values obtained from the symbol table stand in place of the identifiers. They are offsets to a base address (SB) computed at run time, that is, the offsets are added to the base address to yield the effective address. This holds not only for our RISC machine, but is common practice.

10.2. Delayed code generation

Consider as a second example the expression $x + 1$. According to the scheme presented in Section 10.1, we obtain the corresponding code

```

LDW  R0, SB, x      R0 := x
MOV  R1, 1          R1 := 1
ADD  R0, R0, R1     R0 := R0 + R1

```

This shows that the generated code is correct, but certainly not optimal. The flaw lies in the fact that the constant 1 is loaded into a register, although this is unnecessary, because our computer features an instruction which lets constants be added immediately to a register (immediate addressing mode). Evidently some code has been emitted prematurely. The remedy must be to delay code emission in certain cases until it is definitely known that there is no better solution. How is such a delayed code generation to be implemented?

In general, the method consists in associating the information which would have been used for the selection of the emitted code with the resulting syntactic construct. From the principle of attributed grammars presented in Chapter 5, this information is retained in the form of attributes. Code generation therefore depends not only on the syntactically reduced symbols, but also on the values of their attributes. This conceptual extension is reflected by the fact that parser procedures obtain a result parameter which represents these attributes. Because there are usually several attributes, a record structure is chosen for these parameters; we call their type *Item* (Wirth and Gutknecht, 1992).

In the case of our second example, it is necessary to indicate whether the value of a factor, term or expression is held (at run time) in a register, as has been the case so far, or whether the value is a known constant. The latter case will quite likely lead to a later instruction with immediate mode. It now becomes clear that the attribute must indicate the mode in which the factor, term or expression is, that is, where the value is stored and how it is to be accessed. This attribute mode corresponds to the addressing mode of computer instructions, and its range of possible values depends on the set of addressing modes which the target computer features. For each addressing mode offered, there is a corresponding item mode. A mode is also implicitly introduced by object classes. Object classes and item modes partially overlap. In the case of our RISC architecture, there are only three modes:

Item mode	Object class	Addressing mode	Additional attributes
Var	Var	Direct	a Value in memory at address a
Const	Const	Immediate	a Value is the constant a
Reg	-	Register	r Value held in register R[r]

With this in mind, we declare the data type *Item* as a record structure with fields *mode*, *type*, *a* and *r*. Evidently, the type of the item is also an attribute. It will not be mentioned any further below, because we shall consider only the single type *Integer*.

The parser procedures now emerge as functions with result type *Item*. Programming considerations, however, suggest to use proper procedures with a result parameter instead of function procedures.

```
Item = RECORD
    mode: INTEGER;
    type: Type;
    a, r: LONGINT;
END
```

Let us now return to our example to demonstrate the generation of code for the expression $x+1$. The process is shown in Figure 10.1. The transformation of a Var-Item into a Reg-Item is accompanied by the emission of an LDW instruction, and the transformation of a Reg-Item and a Const-Item into a Reg-Item is accompanied by emitting an ADD instruction with immediate operand.

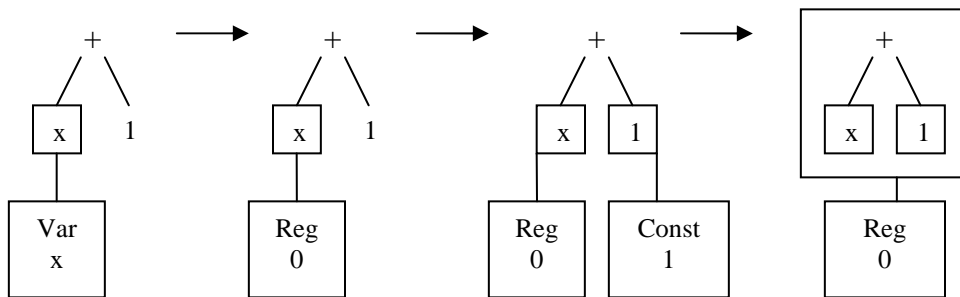


Figure 10.1. Generating items and instructions for the expression $x+1$.

Note the similarity of the two types *Item* and *Object*. Both describe objects, but whereas *Objects* represent declared, named objects, whose visibility reaches beyond the construct of their declaration, *Items* describe objects which are always strictly bound to their syntactic construct. Therefore, it is strongly recommended not to allocate *Items* dynamically (in a heap), but rather to declare them as local parameters and variables.

```
PROCEDURE factor(VAR x: Item);
BEGIN
    IF sym = ident THEN find(obj); Get(sym); x.mode := obj.class; x.a := obj.adr; x.r := 0
    ELSIF sym = int THEN x.mode := Const; x.a := val; Get(sym)
    ELSIF sym = lparen THEN
        Get(sym); expression(x);
        IF sym = rparen THEN Get(sym) ELSE Mark(" ) missing") END
    ELSIF ...
    END
END factor;

PROCEDURE term(VAR x: Item);
    VAR y: Item; op: INTEGER;
BEGIN factor(x);
    WHILE (sym >= times) & (sym <= div) DO
        op := sym; Get(sym); factor(y);
        IF op = times THEN MulOp(x, y) ELSIF op = div THEN DivOp(x, y) END
    END
END term;
```

```

PROCEDURE SimpleExpression(VAR x: Item);
  VAR y: Item; op: INTEGER;
BEGIN
  IF sym = plus THEN Get(sym); term(x)
  ELSIF sym = minus THEN Get(sym); term(x); Neg(x)
  ELSE term(x)
  END ;
  WHILE (sym >= plus) & (sym <= minus) DO
    lop := sym; Get(sym); term(y); AddOp(op, x, y)
  END
END SimpleExpression;

```

```

PROCEDURE Statement;
  VAR obj: Object; x, y: Item;
BEGIN
  IF sym = ident THEN
    find(obj); Get(sym); x.mode := obj.class; x.a := obj.adr; x.r := 0;
    IF sym = becomes THEN Get(sym); expression(y); Store(x, y)
  ELSIF ...
  END
  ELSIF ...
  END
END Statement;

```

The code generating statements are *AddOp*, *Neg*, *MulOp*, *DivOp*, and *Store*. The principle of delayed code emission is also used to avoid the emission of arithmetic instructions if the compiler can perform the operation itself. This is the case when both operands are constants. The technique is known as *constant folding*.

```

PROCEDURE load(VAR x: Item);
  VAR r: INTEGER;
BEGIN (*x.mode # Reg*)
  IF x.mode = Var THEN Put2(Ldw, RH, x.r, x.a)
  ELSIF x.mode = Const THEN Put1(Mov, RH, 0, x.a)
  END ;
  x.mode := Reg; x.r := RH; INC(RH)
END load;

PROCEDURE AddOp(op: INTEGER; VAR x, y: Item);
BEGIN
  IF op = OSS.plus THEN
    IF (x.mode = Const) & (y.mode = Const) THEN x.a := x.a + y.a
    ELSIF y.mode = Const THEN load(x);
      IF y.a # 0 THEN Put1(Add, x.r, x.r, y.a) END
    ELSE load(x); load(y); Put0(Add, RH-2, x.r, y.r); DEC(RH); x.r := RH-1
    END
  ELSE (*op = OSS.minus*)
    IF (x.mode = Const) & (y.mode = Const) THEN x.a := x.a - y.a
    ELSIF y.mode = Const THEN load(x);
      IF y.a # 0 THEN Put1(Sub, x.r, x.r, y.a) END
    ELSE load(x); load(y); Put0(Sub, RH-2, x.r, y.r); DEC(RH); x.r := RH-1
    END
  END
END AddOp;

PROCEDURE MulOp*(VAR x, y: Item); (* x := x * y *)
BEGIN
  IF (x.mode = Const) & (y.mode = Const) THEN x.a := x.a * y.a
  ELSIF (y.mode = Const) & (y.a = 2) THEN load(x); Put1(Lsl, x.r, x.r, 1)
  ELSIF y.mode = Const THEN load(x); Put1(Mul, x.r, x.r, y.a)
  ELSIF x.mode = Const THEN load(y); Put1(Mul, y.r, y.r, x.a); x.mode := Reg; x.r := y.r
  ELSE load(x); load(y); Put0(Mul, RH-2, x.r, y.r); DEC(RH); x.r := RH-1
  END
END MulOp;

```

```

PROCEDURE DivOp*(op: LONGINT; VAR x, y: Item); (* x := x op y *)
BEGIN
  IF op = OSS.div THEN
    IF (x.mode = Const) & (y.mode = Const) THEN
      IF y.a > 0 THEN x.a := x.a DIV y.a ELSE OSS.Mark("bad divisor") END
    ELSIF y.mode = Const THEN
      IF y.a > 0 THEN load(x); Put1(Div, x.r, x.r, y.a) ELSE OSS.Mark("bad divisor") END
      ELSE load(y); load(x); Put0(Div, RH-2, x.r, y.r); DEC(RH); x.r := RH-1
      END
    ELSE (*op = OSS.mod*)
      IF (x.mode = Const) & (y.mode = Const) THEN
        IF y.a > 0 THEN x.a := x.a MOD y.a ELSE OSS.Mark("bad modulus") END
      ELSE load(y);
        load(x); Put0(Div, RH-2, x.r, y.r); Put0(Mov+U, RH-2, 0, 0); DEC(RH); x.r := RH-1
      END
    END
  END DivOp;

```

So far, only arithmetic expressions are treated, and all operands are of type integer. Therefore, no type checking is necessary. Type checking is typically performed along with syntactic analysis. But, whenever arithmetic expressions are evaluated, the inherent danger of overflow exists. The evaluating statements should therefore be suitably guarded. In the case of addition guards can be formulated as follows:

```

IF x.a >= 0 THEN
  IF y.a <= MAX(INTEGER) - x.a THEN x.a := x.a + y.a ELSE Mark("overflow") END
ELSE
  IF y.a >= MIN(INTEGER) - x.a THEN x.a := x.a + y.a ELSE Mark("underflow") END
END

```

The essence of delayed code generation is that code is not emitted before it is clear that no better solution exists. For example, an operand is not loaded into a register before this is known to be unavoidable.

The principle of delayed code generation is also useful in many other cases, but it becomes indispensable when considering computers with complex addressing modes, for which reasonably efficient code has to be generated by making good use of the available complex modes. As an example we consider code emission for a CISC architecture. It typically offers instructions with two operands, one of them also representing the result. Let us consider the expression $u := x + y * z$ and obtain the following instruction sequence:

MOV	y, R0	R0 := y
MUL	z, R0	R0 := R0 * z
ADD	x, R0	R0 := R0 + x
MOV	R0, u	u := R0

This is obtained by delaying the loading of variables until they are to be joined with another operand. Because the instruction replaces the first operand with the operation's result, the operation cannot be performed on the variable's original location, but only on an intermediate location, typically a register. The copy instruction is not issued before this is recognized as unavoidable. A side effect of this measure is that, for example, the simple assignment $x := y$ does not transfer via a register at all, but occurs directly through a copy instruction, which both increases efficiency and decreases code length:

```
MOV y, x      x := y
```

10.3. Indexed variables and record fields

So far we have considered *simple* variables only in expressions and assignments. Access to elements of structured variables, arrays and records, necessitates the selection of the element according to a computed index or a field identifier, respectively. Syntactically, the variable's identifier is followed by one or several selectors. This is mirrored in the parser by a call of the procedure *selector* within *factor* and also in *statement*.

```
find(obj); Get(sym); x.mode := obj.class; x.a := obj.adr; x.r := 0; selector(x)
```

Procedure *selector* processes not only a single selection, but if needed an entire sequence of selections. The following formulation shows that the attribute *type* of the operand *x* is also relevant.

```
PROCEDURE selector(VAR x: Item);
  VAR y: Item; obj: Object;
BEGIN
  WHILE (sym = lbrak) OR (sym = period) DO
    IF sym = lbrak THEN
      Get(sym); expression(y); CheckInt(y);
      IF x.type.form = Array THEN Index(x, y); x.type := x.type.base
      ELSE Mark("not an array")
      END ;
      IF sym = rbrak THEN Get(sym) ELSE Mark("[ ?]") END
    ELSE Get(sym);
      IF sym = ident THEN
        IF x.type.form = Record THEN
          FindField(obj, x.type.fields); Get(sym);
          IF obj # guard THEN Field(x, obj); x.type := obj.type
          ELSE Mark("undef")
          END
        ELSE Mark("not a record")
        END
      ELSE Mark("ident?")
      END
    END
  END
END selector;
```

The address of the selected element is given by the formulas derived in Section 8.3. In the case of a field identifier the address computation is performed by the compiler. The address is the sum of the variable's address and the field's offset.

```
PROCEDURE Field(VAR x: Item; y: Object); (* x := x.y *)
BEGIN x.a := x.a + y.val; x.type := y.type
END Field;
```

In the case of an indexed variable, code is emitted according to the formula

$$\text{adr}(a[k]) = \text{adr}(a) + k * \text{size}(T)$$

Here *a* denotes the array variable, *k* the index, and *T* the type of the array's elements. An index computation requires two instructions; the scaled index is added to the register component of the address. Let the index be stored in register *R.j*, and let the array address be stored in register *R.i*.

```
MUL R.j, R.j, size(T)
ADD R.i, R.i, R.j
```

Procedure *Index* emits the above index code, checks whether the indexed variable is indeed an array, and computes the element's address directly if the index is a constant. Here we notice that a new item mode is necessary to represent the case of an indexed variable. Its location is determined by an offset (*x.a*) and a register (*x.r*) holding the index. We call this mode *RegI* (for register indirect). In contrast to modes *Var* and *Const* it does not stem from a declared object, but emerges during evaluation (of an indexed variable).

```
PROCEDURE Index(VAR x, y: Item); (* x := x[y] *)
  VAR z: Item;
BEGIN
  IF y.mode = Const THEN
    IF (y.a < 0) OR (y.a >= x.type.len) THEN Mark("index out of range") END ;
    x.a := x.a + y.a * x.type.base.size
```

```

ELSE
  IF y.mode # Reg THEN load(y) END ;
  Put1(Mul, y.r, y.r, x.type.base.size);
  IF x.mode = Var THEN Put0(Add, y.r, x.r, y.r); x.r := y.r; x.mode := Regl
    ELSIF x.mode = Regl THEN Put0(Add, x.r, x.r, y.r); DEC(RH)
  END
END
END Index;

```

In fact, also in mode *Var* an address register is involved, namely one that holds a base address (SB). The difference between *Var* and *Regl* modes is that in the former the base register *x.r* must not be overwritten. We can now show the code resulting from the following program fragment containing one- and two-dimensional arrays.

```

VAR i, j: INTEGER;                                adr 0, 4
  a: ARRAY 4 OF INTEGER;                          adr 8
  b: ARRAY 3 OF ARRAY 5 OF INTEGER;              adr 24

LDW  R0, SB, 4          i := a[j]
MUL  R0, R0, 4
ADD  R0, SB, R0
LDW  R0, SB, 8          a
STW  R0, SB, 0          i

LDW  R0, SB, 16         i := a[2]
STW  R0, SB, 0

LDW  R0, SB, 0          i := a[i+j];
LDW  R1, SB, 4
ADD  R0, R0, R1         i+j
MUL  R0, R0, 4
ADD  R0, SB, R0
LDW  R0, R0, 8
STW  R0, SB, 0          i

LDW  R0, SB, 0          i := b[i][j]
MUL  R0, R0, 20
ADD  R0, SB, R0
LDW  R1, SB, 4          j
MULI R1, R1, 4
ADD  R0, R0, R1
LDW  R0, R0, 24         b
STW  R0, SB, 0          i

LDW  R0, SB, 80         i := b[2][4]
STW  R0, SB, 0

LDW  R0, SB, 0          i := a[a[i]]
MUL  R0, R0, 4
ADD  R0, SB, R0
LDW  R0, R0, 8
MUL  R0, R0, 4
ADD  R0, SB, R0
LDW  R0, R0, 8
STW  R0, SB, 0

```

Note that the validity of the index can be checked only if the index is a constant, that is, it is of known value. Otherwise the index cannot be checked until run time. Although the test is of course redundant in correct programs, its omission is not recommended. In order to safeguard the abstraction of the array structure the test is wholly justified. However, the compiler designer should attempt to achieve utmost efficiency. The test takes the form of the statement

```
IF (k < 0) OR (k >= n) THEN HALT END
```

where k is the index and n the array's length. With our RISC, this is implemented by two instructions only, a comparison and a trap. The comparison takes advantage of unsigned arithmetic, where negative values are interpreted as large integers. The trap is represented by a conditional branch and link instruction. The branch condition is HI = 6, its destination is a fixed location of a trap routine.

```
CMP    I, k
BL     HI, trap
```

Procedure *Index* is extended accordingly:

```
IF y.mode # Reg THEN load(y) END ;
Put1(Cmp, 0, y.r, x.type.base.len);
Put3(15, 6, trap-pc);
```

Finally, an example of a program is shown with nested data structures. It demonstrates clearly how the special treatment of constants in selectors simplifies the code for address computations. Compare the code resulting for variables indexed by expressions with those indexed by constants. Index range checks have been omitted for the sake of brevity.

```
TYPE R0 = RECORD x, y: INTEGER END ;
R1 = RECORD u: INTEGER;          offset 0
      v: ARRAY 4 OF R0;         offset 4
      w: INTEGER                offset 36
END ;
VAR i, j, k: INTEGER;           adr 0, 4, 8
s: ARRAY 2 OF R1;              adr 12

LDW    R0, SB, 0      k := s[i].u
MUL    R0, R0, 40
ADD    R0, SB, R0
LDW    R0, R0, 12    s[i].u
STW    R0, SB, 8      k

LDW    R0, SB, 88    k := s[1].w
STW    R0, SB, 8

LDW    R0, SB, 0      k := s[i].v[j].x
MUL    R0, R0, 40
ADD    R0, SB, R0
LDW    R1, SB, 4      j
MUL    R1, R1, 8
ADD    R0, R0, R1
LDW    R0, R0, 16    s[i].v[j].x
STW    R0, SB, 8

LDW    R0, SB, 76    k := s[1].v[2].y
STW    R0, SB, 8

LDW    R0, SB, 0      s[0].v[i].y := k
MUL    R0, R0, 8
ADD    R0, SB, R0
LDW    R1, SB, 8      k
STW    R1, R0, 20
```

The desire to keep target-dependent parts of the compiler separated from target-independent parts suggests that code generating statements should be collected in the form of procedures in a separate module. We shall call this module OSG and present its interface. It contains several of the generator procedures encountered so far. The others will be explained in Chapters 11 and 12.

```
DEFINITION OSG;
IMPORT OSS;
```

```

CONST Head = 0; Var = 1; Par = 2; Const = 3; Fld = 4; Typ = 5
  SProc = 6; SFunc = 7; Proc = 8;
  Boolean = 0; Integer = 1; Array = 2; Record = 3;

TYPE Object = POINTER TO ObjDesc;
  ObjDesc = RECORD
    class, lev: INTEGER;
    next, dsc: Object;
    type: Type;
    name: OSS.Ident;
    val: LONGINT;

  END ;

  Type = POINTER TO TypeDesc;
  TypeDesc = RECORD
    form: INTEGER;
    fields: Object;
    base: Type;
    size, len: INTEGER;

  END ;

  Item = RECORD
    mode, lev: INTEGER;
    type: Type;
    a: LONGINT;

  END ;

VAR boolType, intType: Type;
  curlev, pc: INTEGER;

PROCEDURE FixLink (L: LONGINT);
PROCEDURE IncLevel (n: INTEGER);
PROCEDURE MakeConstItem (VAR x: Item; typ: Type; val: LONGINT);
PROCEDURE MakeItem (VAR x: Item; y: Object);
PROCEDURE Field (VAR x: Item; y: Object);
PROCEDURE Index (VAR x, y: Item);
PROCEDURE Neg (VAR x: Item);
PROCEDURE AddOp(op: INTEGER; VAR x, y: Item);
PROCEDURE MulOp(VAR x, y: Item);
PROCEDURE DivOp(op: INTEGER;VAR x, y: Item);
PROCEDURE Relation (op: INTEGER; VAR x, y: Item);
PROCEDURE Store (VAR x, y: Item);

PROCEDURE Parameter (VAR x: Item; ftyp: Type; class: INTEGER);
PROCEDURE CJump (VAR x: Item);
PROCEDURE BJump (L: LONGINT);
PROCEDURE FJump (VAR L: LONGINT);
PROCEDURE Call (VAR x: Item);
PROCEDURE Enter (size: LONGINT);
PROCEDURE Return (size: LONGINT);
PROCEDURE Open;
PROCEDURE Header (size: LONGINT);
PROCEDURE Close;
END OSG.

```

10.4. Exercises

- 10.1. Improve the Oberon-0 compiler in such a way that multiplication and division instructions are replaced by efficient shift and mask instructions, if a factor or the divisor is a power of 2.
- 10.2. Had the assignment statement in Oberon been defined in a form where the assigned expression occurs to the left of the variable, that is for example by the form $e =: v$, would compilation of assignments be simpler in any way?

- 10.3. Consider the introduction of a multiple assignment in Oberon of the form $e := x_0 := x_1 := \dots := x_n$. Implement it. Does the definition of its semantics present any problems?
- 10.4. Change the definition of expressions in Oberon to that of Algol 60 (see Exercise 2.1) and implement the changes. Discuss the advantages and disadvantages of the two forms.

11. Conditional and Repeated Statements and Boolean Expressions

11.1. Comparisons and jumps

Conditional and repeated statements are implemented with the aid of branch instructions. As a first example, let us consider the simplest form of conditional statement:

```
IF x = y THEN StatSequence END
```

A possible mapping into a sequence of instructions is straightforward:

```
IF x = y          EQL x, y
                  BF  L
THEN StatSequence code(StatSequence)
END              L  ...
```

Our considerations are based once again on a stack architecture. Instruction EQL tests the two operands for equality and replaces them on the stack by the Boolean result. The subsequent branch instruction BF (branch if FALSE) leads to the destination label L if this result is FALSE, and removes it from the stack. Similarly to EQL, conditional branch instructions are postulated for all other relations.

Unfortunately, however, such compiler-friendly computers are hardly widespread. Rather more common are computers whose branch instructions depend on the comparison of a register value with 0. We denote them as BNE (branch if not equal), BLT (branch if less than), BGE (branch if greater or equal), BLE (branch if less or equal), and BGT (branch if greater than). The code sequence corresponding to the above example is

```
IF x = y          SUB R0, x, y
                  BNE R0, L
THEN StatSequence code(StatSequence)
END              L  ...
```

In order to compare two numbers, subtraction is used. A specific comparison instruction is superfluous. However, we will use the mnemonic CMP, whenever the subtraction is used for comparison only. The relevant result is deposited in flag registers called *condition codes*. There are four flags denoted by N, Z, C, and V, indicating whether the difference is negative or zero respectively. C represents the carry out bit, and V indicates overflow of signed numbers. All conditional branch instructions implicitly test this register as argument. In fact, typically all register instructions have these side-effects on the flags.

```
IF x = y          CMP x, y
                  BNE L
THEN StatSequence code(StatSequence)
END              L  ...
```

11.2. Conditional and repeated statements

The question of how a Boolean value is to be represented by an item now arises. In the case of a stack architecture the answer is easy: since the result of a comparison lies on the stack like any other result, no special item mode is necessary. The presence of flags, however, requires further thought. We shall first restrict our consideration to the simple cases of pure comparisons without further Boolean operators.

In the case of an architecture with a flag scheme, it is necessary to indicate in the resulting item which relation is specified by the comparison. For the latter a new attribute is required; we call the new mode *Cond* and its new attribute (record field) *r*. The mapping of relations to values of *r* is defined by

=	1	#	9
<	5	>=	13
<=	6	>	14

The construct containing comparisons is the expression. Its syntax is

expression = SimpleExpression [("=" | "#" | "<" | "<=" | ">" | ">=") SimpleExpression].

The corresponding, extended parser procedure is easily derived:

```

PROCEDURE expression(VAR x: Item);
  VAR y: Item; op: INTEGER;
BEGIN SimpleExpression(x);
  IF (sym >= eql) & (sym <= geq) THEN
    op := sym; Get(sym); SimpleExpression(y); Relation(op, x, y)
  END
  x.type := BoolType
END expression;

PROCEDURE Relation(op: INTEGER; VAR x, y: Item);
BEGIN
  IF y.mode = Const THEN load(x); Put1(Cmp, x.r, x.r, y.a); DEC(RH)
  ELSE load(x); load(y); Put0(Cmp, x.r, x.r, y.r); DEC(RH, 2)
  END ;
  SetCC(x, relmap[op - OSS.eql])
END Relation;

PROCEDURE SetCC(VAR x: Item; n: LONGINT);
BEGIN x.mode := Cond; x.a := 0; x.b := 0; x.r := n
END SetCC;

```

The code scheme presented at the beginning of this chapter yields the corresponding parser program for handling the IF construct in *StatSequence* in its simplified form (without ELSE and ELSIF).

```

ELSIF sym = if THEN
  Get(sym); expression(x); CFJump(x);
  IF sym = then THEN Get(sym) ELSE Mark("THEN ?") END ;
  StatSequence; Fixup(x.a)
  IF sym = end THEN Get(sym) ELSE Mark("END ?") END

```

Procedure *CFJump(x)* generates the necessary branch instruction according to its parameter *x.rc* in such a way that the jump is taken if the specified condition is not satisfied.

Here a difficulty becomes apparent which is inherent in all single-pass compilers. The destination location of branches is still unknown when the instruction is to be emitted. This problem is solved by adding the location of the branch instruction as an attribute to the item generated. This attribute is used later when the destination of the jump becomes known in order to complete the branch with its true address. This is called a *fixup*. The simple solution is possible only if code is deposited in a global array where elements are accessible at any time. It is not applicable if the emitted code is immediately stored on disk. To represent the address of the incomplete branch instruction we use the item field *a*.

```

PROCEDURE CFJump(VAR x: Item); (*conditional branch forward*)
BEGIN Put3(2, negated(x.r), x.a); x.a := pc-1
END CFJump;

PROCEDURE negated(cond: LONGINT): LONGINT;
BEGIN
  IF cond < 8 THEN RETURN cond + 8 ELSE RETURN cond - 8 END
END negated;

PROCEDURE Fixup(L: LONGINT);
BEGIN code[L] := code[L] DIV 1000000H * 1000000H + (pc - L-1) MOD 1000000H
END Fixup;

```

Note that branch instructions use addresses relative to the instruction's location (PC-relative); therefore the value $pc-L-1$ is used.

Finally, we have to show how conditional statements in their general form are compiled; the syntax is

```
"IF" expression "THEN" StatSequence
{"ELSIF" expression "THEN" StatSequence}
["ELSE" StatSequence]
"END"
```

and the corresponding code pattern is

```
IF expression THEN          code(expression)
                             Bcond L0
    StatSequence             code(StatSequence)
                             BR L
ELSIF expression THEN      L0 code(expression)
                             Bcond L1
    StatSequence             code(StatSequence)
                             BR L
ELSIF expression THEN      L1 code(expression)
                             Bcond L2
    StatSequence             code(StatSequence)
                             BR L
.....
ELSE StatSequence          Ln code(StatSequence)
END                          L ...
```

from which the parser statements can be derived as part of procedure *StatSequence*. Although an arbitrary number of ELSIF constructs can occur and thereby also an arbitrary number of jump destinations $L1, L2, \dots$ may result, a single item variable x suffices. It is assigned a new value for every ELSIF instance.

```
ELSIF sym = if THEN
  Get(sym); expression(x); CFJump(x);
  IF sym = then THEN Get(sym) ELSE Mark("THEN ?") END ;
  StatSequence; L := 0;
  WHILE sym = elsif DO
    Get(sym); FJump(L); Fixup(x.a); expression(x); CFJump(x);
    IF sym = then THEN Get(sym) ELSE Mark("THEN ?") END ;
    StatSequence
  END ;
  IF sym = else THEN Get(sym); FJump(L); Fixup(x.a); StatSequence
  ELSE Fixup(x.a)
  END ;
  Fixup(L);
  IF sym = end THEN Get(sym) ELSE Mark("END ?") END
  ...

PROCEDURE FJump(VAR L: LONGINT); (*unconditional branch forward*)
BEGIN Put3(2, 7, L); L := pc-1
END FJump
```

However, a new situation arises in which not only a single branch refers to the destination label L at the end, but an entire set, namely as many as there are IF and ELSIF branches in the statement. The problem is elegantly solved by storing the links of the list of incomplete branch instructions in these instructions themselves, and to let variable L represent the root of this list. The links are established by the parameter of the *Put* operation called in *FJump*. It suffices to replace procedure *Fixup* by *FixLink*, in which the entire list of instructions to be fixed up is traversed. It is essential that variable L is declared local to the parser procedure *StatSequence*,

because statements may be nested, which leads to recursive activation. In this case, several instances of variable L coexist representing different lists.

```

PROCEDURE FixLink(L: LONGINT);
  VAR L1: LONGINT;
BEGIN
  WHILE L # 0 DO
    L1 := code[L] MOD 10000H; Fixup(L); L := L1
  END
END FixLink;

```

Compilation of the WHILE statement is very similar to that of the simple IF statement. In addition to the conditional forward jump, an unconditional backward jump is necessary. The syntax and the corresponding code pattern are:

WHILE expression DO	L0 code(expression)
	Bcond L1
StatSequence	code(StatSequence)
END	BR L0
	L1 ...

From this we derive the corresponding, extended parser procedure:

```

ELSIF sym = while THEN
  Get(sym); L := pc; expression(x); CFJump(x);
  IF sym = do THEN Get(sym) ELSE Mark("DO ?") END ;
  StatSequence; BJump(L); FixLink(x.a);
  IF sym = end THEN Get(sym) ELSE Mark("END ?") END

PROCEDURE BJump(L: LONGINT); (*unconditional backward jump*)
BEGIN Put3(2, 7, L-pc-1)
END BJump;

```

Even simpler is the compilation of repeat statements. The corresponding parsing section is

```

ELSIF sym = repeat THEN
  Get(sym); L := pc; StatSequence;
  IF sym = until THEN Get(sym); expression(x); CBJump(x, L)
  ELSE OSS.Mark("UNTIL ?"); Get(sym)
  END

PROCEDURE CBJump*(VAR x: Item; L: LONGINT); (*conditional backward jump*)
BEGIN Put3(2, negated(x.r), L-pc-1)
END CBJump;

```

To summarize, we display two statements using variables i and j , together with the generated code:

```

IF i < j THEN i := 0 ELSIF i = j THEN i := 1 ELSE i := 2 END ;
WHILE i > 0 DO i := i - 1 END ;
REPEAT i := i - 1 UNTIL i = 0

0 LDW R0, SB, 4 i
1 LDW R1, SB, 8 j
2 CMP R0, R1
3 BGE 3 (jump over 3 instructions to 7)
4 MOV R0, 0
5 STW R0, SB, 4 i := 0
6 B 9
7 LDW R0, SB, 4 i
8 LDW R1, SB, 8 j
9 CMP R0, R1
10 BNE 3 (jump over 3 instructions to 14)
11 MOV R0, 1
12 STW R0, SB, 4 i := 1

```

```

13  B      2
14  MOV   R0, 2
15  STW   R0, SB, 4      i := 2
16  LDW   R0, SB, 4
17  CMP   R0, R0
18  BLE   4              (jump over 4 instructions to 23)
19  LDW   R0, SB, 4
20  SUB   R0, R0, 1
21  STW   R0, SB, 4      i := i - 1
22  BEQ   -7            (jump back over 7 instructions to 16)
23  LDW   R0, SB, 4      i
24  SUB   R0, R0, 1
25  STW   R0, SB, 4
26  LDW   R0, SB, 0
27  CMP   R0, 0
28  BNE   -6            (jump back over 6 instructions to 23)

```

11.3. Boolean operations

It is of course tempting to treat Boolean expressions in the same way as arithmetic expressions. Unfortunately, however, this would in many cases lead not only to inefficient, but even to wrong code. The reason lies in the definition of Boolean operators, namely

```

p OR q  = if p then TRUE else q
p & q   = if p then q else FALSE

```

This definition specifies that the second operand q need not be evaluated if the result is uniquely given by the value of the first operand p . Programming language definitions even go a step further by specifying that in these cases the second operand *must* not be evaluated. This rule is postulated in order that the second operand may be left undefined without causing program execution to be terminated. A frequent example involving a pointer x is

```
(x # NIL) & (x^.size > 4)
```

Here $x^.size$ is undefined, if $x = \text{NIL}$. Boolean expressions with Boolean operators therefore assume the form of conditional statements (more precisely, conditional expressions), and it is appropriate to use the same compilation techniques as for conditional statements. Boolean expressions and conditional statements merge, as the following example shows. The statement

```
IF (x <= y) & (y < z) THEN S END
```

is compiled in the same way as its equivalent formulation

```
IF x <= y THEN IF y < z THEN S END END
```

With the intention of deriving a suitable code pattern, let us first consider the following expression containing three relations connected by the $\&$ operator. We postulate the desired code pattern as shown below, considering only the pattern to the left for the moment. a, b, \dots, f denote numeric values. The labels T and F denote the destinations for the cases when the expression is true or false, respectively.

```
(a < b) & (c < d) & (e < f)
```

```

CMP a, b          CMP a, b
BGE F            BGE F
CMP c, d          CMP c, d
BGE F            BGE F
CMP e, f          CMP e, f
BGE F            BLT T
(T)              (F)

```

As the left hand pattern shows, a conditional branch instruction is emitted for every & operator. The jump is executed if the preceding condition is not satisfied (F-jump). This results in the instructions BGE to represent the < relation, BNE for the = relation, and so on.

If we consider the problem of generating the required code, we can see that the parser procedure term, as it is known for processing arithmetic terms, must be extended slightly. In particular, a branch instruction must be emitted before the second operand is processed, whereas at the end this instruction's address must be fixed up. The former task is performed by procedure *Op1*, the latter by *Op2*.

```

PROCEDURE term(VAR x: Item);
  VAR y: Item; op: INTEGER;
BEGIN factor(x);
  WHILE (sym >= times) & (sym <= and) DO
    op := sym; Get(sym);
    IF op = times THEN factor(y); MulOp(x, y)
    ELSIF (op = div) OR (op = mod) THEN factor(y); DivOp(op, x, y)
    ELSE op = and THEN And(x); factor(y); And2(x, y)
  END
END term;

PROCEDURE And1(VAR x: Item); (* x := x & *)
BEGIN
  IF x.mode # Cond THEN loadCond(x) END ;
  Put3(BC, negated(x.r), x.a); x.a := pc-1; FixLink(x.b); x.b := 0
END And1;

PROCEDURE And2(VAR x, y: Item); (* x := x & y *)
BEGIN
  IF y.mode # Cond THEN loadCond(y) END ;
  x.a := merged(y.a, x.a); x.b := y.b; x.r := y.r
END And2;

```

If the first Boolean factor is represented by item *x* in mode *Cond*, then at the present position *x* is TRUE and the instructions for the evaluation of the second operand must follow. They must be skipped, if the condition is FALSE. However, if item *x* is not in mode *Cond*, it must be converted into this mode. This task is executed by procedure *loadCond*. We assume that the value FALSE is represented by 0. The attribute value *c* = 1 therefore causes the instruction BEQ to become active, if *x* equals 0.

```

PROCEDURE loadCond(VAR x: Item);
BEGIN
  IF x.type.form = Const THEN x.r := 15 - 8*x.a;
  ELSE Put1(Cmp, 0, x.r, 0); x.r := 3; DEC(RH)
  END ;
  x.mode := Cond
END loadCond;

```

The OR operator is treated analogously, with the difference that jumps are taken if their respective conditions are satisfied (T-jump). The instructions are listed in a second list with links in the item field *b*. Consider again the left-hand code pattern only:

(a < b) OR (c < d) OR (e < f)	
CMP a, b	CMP a, b
BLT T	BLT T
CMP c, d	CMP c, d
BLT T	BLT T
CMP e, f	CMP e, f
BLT T	BGE F
(F)	(T)

Next, we consider the implementation of negation. Here it turns out that under the scheme presented no instructions need be emitted whatsoever. Only the condition value represented by

the item field *c* has to be negated, and the lists of F-jumps and T-jumps need be exchanged. The result of negation is shown in the code patterns in Figures 11.1 and 11.2 on the right-hand side for both expressions with & and OR operators. The affected procedures are extended as shown below:

```

PROCEDURE SimpleExpression(VAR x: Item);
  VAR y: Item; op: INTEGER;
BEGIN term(x);
  WHILE (sym >= plus) & (sym <= or) DO
    op := sym; Get(sym);
    IF op = or THEN Or1(x); term(y); Or2(x, y)
    ELSE term(y); AddOp(op, x, y)
  END
END
END SimpleExpression;

PROCEDURE Or1(VAR x: Item); (* x := x OR *)
BEGIN
  IF x.mode # Cond THEN loadCond(x) END ;
  Put3(BC, x.r, x.b); x.b := pc-1; FixLink(x.a); x.a := 0
END Or1;

PROCEDURE Or2(VAR x, y: Item);
BEGIN
  IF y.mode # Cond THEN loadCond(y) END ;
  x.a := y.a; x.b := merged(y.b, x.b); x.r := y.r
END Or2;

```

When compiling expressions with & and OR operators, care must be taken that in front of every & condition *P*, and in front of every OR condition $\sim P$, must hold. The respective lists of jump instructions must be traversed (the T-list for &, the F-list for OR), and the designated instructions must be fixed up appropriately. This occurs through procedure calls of *FixLink* in *Op1*. As examples, we consider the expressions

(a < b) & (c < d) OR ((e < f) & (g < h))
 (a < b) OR (c < d) & ((e < f) OR (g < h))

and the resulting codes:

<pre> CMP a, b BGE F0 CMP c, d BLT T F0 CMP e, f BGE F CMP g, h BGE F (T) </pre>	<pre> CMP a, b BLT T0 CMP c, d BGE F T0 CMP e, f BLT T CMP g, h BGE F (T) </pre>
---	---

It may also happen that a list of a subordinate expression may merge with the list of its containing expression (see F-link in the pattern for Q).. This merger is accomplished by procedure *merged(a, b)*, yielding as its value the concatenation of its argument lists.

11.4. Assignments to Boolean variables

Compilation of an assignment to a Boolean variable *q* is certainly more complicated than commonly expected. The reason is the item mode *Cond*, which must be converted into an assignable value 0 or 1. This is achieved by the following code pattern:

```

T    MOV R0, 1
     B L
F    MOV R0, 0

```

```
L    STW    R0, SB, q
```

This causes the simple assignment $q := x < y$ to appear as a disappointingly long code sequence. We should, however, be aware that Boolean variables (commonly called *flags*) occur (should occur) infrequently, although the notion of the type *Boolean* is indeed fundamental. It is inappropriate to strive for optimal implementation of rarely occurring constructs at the price of an intricate process. On the other hand, it is essential that the frequent cases are handled optimally.

Nevertheless, we handle assignments of a Boolean item not in the *Cond* mode as a special case, namely as a conventional assignment avoiding the involvement of jumps. Hence, the assignment $p := q$ results in the expected code sequence

```
LDW    R0, SB, q
STW    R0, SB, p
```

As a consequence, the procedures *load* and *Store* turn out as follows (see also Ch. 10):

```
PROCEDURE load(VAR x: Item);
BEGIN
  IF x.mode # Reg THEN
    IF x.mode = Var THEN Put2(Ldw, R, x.r, x.a); INC(R)
    .....
    ELSIF x.mode = RegI THEN Put2(Ldw, R, x.r, x.a)
    ELSIF x.mode = Cond THEN
      Put3(2, negated(x.r), 2);
      FixLink(x.b); Put1(Mov, R, 0, 1); Put3(2, 7, 1);
      FixLink(x.a); Put1(Mov, R, 0, 0); incR
    END ;
    x.mode := Reg; x.r := RH-1; x.a := 0; x.b := 0
  END
END load;

PROCEDURE Store(VAR x, y: Item); (* x := y *)
BEGIN ...
  IF y.mode # Reg THEN load(y) END ;
  IF x.mode = Var THEN Put2(Stw, y.r, x.r, x.a); DEC(R)
  ELSIF x.mode = RegI THEN Put2(Stw, y.r, x.r, x.a); DEC(RH, 2)
  ELSE Mark("illegal assignment")
  END
END Store;
```

11.5. Exercises

11.1. Mutate the language Oberon-0 into a variant Oberon-D by redefining the conditional and the repeated statement as follows:

```
statement = ...
  "IF" guardedStatements {"|" guardedStatements} "FI" |
  "DO" guardedStatements {"|" guardedStatements} "OD" .
guardedStatements = condition "." statement {";" statement} .
```

The new form of statement

```
IF B0 . S0 | B1 . S1 | ... | Bn . Sn FI
```

shall mean that of all conditions (Boolean expressions) B_i that are true, one is selected arbitrarily and its corresponding statement sequence S_i is executed. If none is true, program execution is aborted. Any statement sequence S_i will be executed only when the corresponding condition B_i is true. B_i is therefore said to be the guard of S_i .

The statement

DO $B_0 . S_0$ | $B_1 . S_1$ | ... | $B_n . S_n$ OD

shall mean that as long as any of the conditions B_i is true, one of them is chosen arbitrarily, and its corresponding statement sequence S_i is executed. The process terminates as soon as all B_i are false. Here too, the B_i function as guards. The DO-OD construct is a repetitive, nondeterministic construct. Adjust the compiler accordingly.

11.2. Extend Oberon-0 and its compiler by a FOR statement:

```
statement = [assignment | ProcedureCall |  
             IfStatement | WhileStatement | ForStatement.  
ForStatement = "FOR" identifier ":=" expression "TO" expression ["BY" expression]  
              "DO" StatementSequence "END" .
```

The expression preceding the symbol TO specifies the starting value, the one thereafter the ending value of the control variable denoted by the identifier. The expression after BY indicates the increment. If missing, let 1 be its default value.

11.3. Consider the implementation of the case statement of Oberon. Its essential property is that it uses a table of jump addresses for the various cases, and an indexed jump instruction.

12. Procedures and the Concept of Locality

12.1. Run-time organization of the store

Procedures, which are also known as subroutines, are perhaps the most important tool for structuring programs. Because of their frequency of occurrence, it is mandatory that their implementation is efficient. Implementation is based on the branch instruction which saves the current PC value and thereby the point of return after termination of the procedure, when this value is reloaded into the PC register.

The question as to where the return address should be saved arises immediately. In many computers it is deposited in a register (here called LNK), and we have adopted this solution in our RISC. A call is therefore implemented by a single branch and link (BL) instruction, and the return at the end of the procedure by a single branch with LNK as source. This guarantees the utmost efficiency, because no additional memory access is involved. But having to save the register's value into memory before the next procedure call is unavoidable, because otherwise the old return address would be overwritten. Thereby the return address of the first call would be lost. In the implementation of a compiler this *link register* value must be saved at the beginning of each procedure call.

To store the link, a stack is the obvious solution. The reason is that procedure activations occur in a nested fashion; procedures terminate in the reverse order of their calls. The store for the return addresses must therefore operate according to the *first-in last-out* principle. This results in the following, fixed code sequences at the beginning and end of every procedure. They are called the procedure's *prologue* and *epilogue*. Here we will use R14 for the stack pointer SP

Call		BL	P	branch and link to subroutine
Prologue	P	SUB	SP, SP, 4	
		STW	LNK, SP, 0	push link
Epilogue		LDW	LNK, SP, 0	pop link
		ADD	SP, SP, 4	
		B	LNK	return jump

This code pattern is valid under the assumption that the BL instruction deposits the return address in register LNK (here R15). Note that this is specified as a hardware feature (Chapter 9), whereas the use of R14 as stack pointer is merely a software convention determined by the compiler design or by the underlying operating system. Whenever the system is started, R14 must be initialized to point to an area of memory reserved for the stack.

Algol 60 introduced the very fundamental concept of *local* variables. It implied that every identifier declared had a limited range of visibility and validity. Adopted in Pascal and also in Oberon, this range is the procedure body. In concrete terms, variables may be declared local to a procedure such that they are visible and valid within this procedure only. The intended consequence is that upon entry to the procedure memory is allocated automatically for these local variables, and it is released upon the procedure's termination. Local variables of different procedures may therefore share the same storage area, but never simultaneously, of course.

At first sight this scheme seems to inflict a certain loss of efficiency upon the procedure call mechanism. Fortunately, however, this need not be so, because the storage blocks for the sets of local variables can be allocated, like return addresses, according to the stack principle. The return address may indeed also be considered as a (hidden) local variable, and it is only natural to use the same stack for variables and return addresses. The stack pointer is incremented by the size of the variable block instead of by 4. The storage blocks are called *procedure activation records* or *activation frames*. Release of a block upon procedure termination is achieved by simply resetting the stack pointer to its value before the procedure call. Hence, allocation and release of local storage is optimally efficient.

Addresses of local variables generated by the compiler are always relative to the base address of the respective activation frame. Since in programs most variables are local, their addressing also must be highly efficient. This is achieved by reserving a register to hold the base address, and to make use of the fact that the effective address is the sum of a register value and the instruction's address field (register relative addressing mode). The reserved register is called the *frame pointer* (FP). This scheme makes it possible to call procedures recursively. These considerations are taken into account by the following prologue and epilogue:

Prolog	P	SUB	SP, SP, n+8	SP := SP-(n+8) (n = frame size)
		STW	LNK, SP, n+4	push link
		STW	FP, SP, n	push FP
		ADD	FP, SP, n	FP := SP+n
Epilog		LDW	LNK, FP, 4	pop FP
		LDW	FP, FP, -8	pop link
		MOV	SP, FP	SP := FP
		B	LNK	return jump

The activation frames resulting from consecutive procedure calls are linked by a list of their base addresses. The list is called the *dynamic link*, because it denotes the dynamic sequence of procedure activations. Its root lies in the frame pointer register FP (see Figure 12.1).

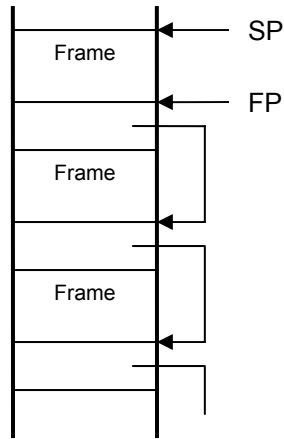


Figure 12.1. List of activation frames in the stack.

The state of the stack before and after a procedure call is shown in Figure 12.2. Note that the epilogue reverts the stack to its original state by removing return address and dynamic link entry.

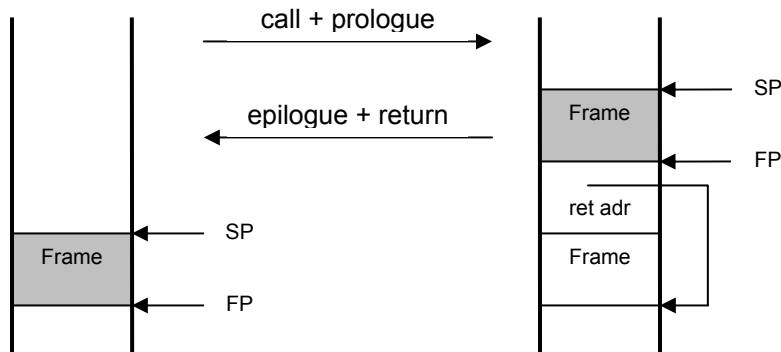


Figure 12.2. States of the stack before and after procedure call.

12.2. Addressing of variables

We recall that the address of a local variable is relative to the base address of the activation frame containing the variable, and that this base address is held in register FP. The latter, however, holds only for the record activated last, and thereby only for variables which belong to the procedure in which they are referenced. In many programming languages procedure declarations may be nested, giving rise to references to variables which are local to some procedure, but not to the procedure referencing them. The following example demonstrates the situation, with R being local to Q, and Q and S local to P:

	Object	Level
PROCEDURE P;	P	0
VAR x: INTEGER;	x	1
PROCEDURE Q;	Q	1
VAR y: INTEGER;	y	2
PROCEDURE R;	R	2
VAR z: INTEGER;	z	3
BEGIN x := y + z		
END R;		
BEGIN R		
END Q ;		
PROCEDURE S;	S	1
BEGIN Q		
END S;		
BEGIN Q; S		
END P;		

Let us trace the chain of calls $P \rightarrow Q \rightarrow R$. It is tempting to believe that, when accessing variables x , y , or z in R , their base address could be obtained by traversing the dynamic link list. The number of steps would be the difference between the levels of the call and of the declaration. This difference is 2 for x , 1 for y , and 0 for z . But this assumption is wrong. R could also be reached through the call sequence $P \rightarrow S \rightarrow Q \rightarrow R$ as shown in Figure 12.3. Access to x would then lead in two steps to the activation frame of S instead of P .

Evidently, a second list of activation records is necessary which mirrors the static order of nesting rather than the dynamic order of calls. Hence a second link must be established upon every procedure call. The so-called *procedure mark* now contains, in addition to the return address and the dynamic link, a *static link* element. The static link of a procedure P points to the activation record of the procedure which contains P , that is, in which P is declared locally. It should be noted that this pointer is superfluous for procedures declared globally, if global variables are addressed directly, that is, without base address. Since this is typically the case, and since most procedures are declared globally, the additional complexity caused by the static chain is acceptable. With some justification the absolute addressing of global variables can be considered as a special case of local variable addressing leading to an increase in efficiency.

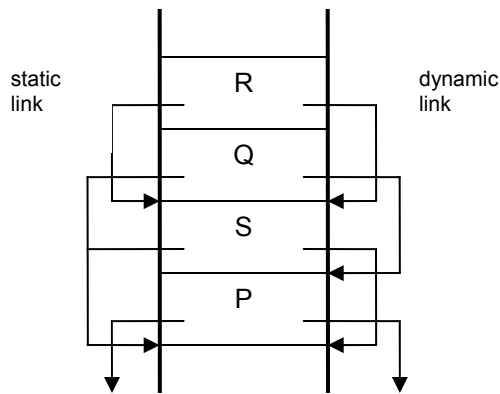


Figure 12.3. Dynamic and static links in the stack.

Finally, note that access to variables via the static link list (intermediate level variables) is less efficient than access to strictly local variables, because every step through the list requires an additional memory access. Several solutions have been proposed and implemented to eliminate this loss of efficiency. They ultimately always rely on the mapping of the static list onto a set of base registers. We consider this as an optimization at the wrong place. First, registers are scarce resources which should not be given away too easily. And second, the copying of link elements into registers upon every call and return may easily cost more than it saves, in particular because references to intermediate-level variables occur quite rarely in practice. The optimization may therefore turn out to be quite the reverse.

In fact, programming experience has shown that the access of intermediate level variables is bad practice and better be avoided. Implementers must welcome this insight, as it makes the static link superfluous. If in addition we renounce the allocation on the stack of objects whose size is unknown at compile-time (such as dynamic array parameters in Algol), we can even omit the dynamic link. This because the amounts by which the stack pointer is to be decreased and increased are known by the compiler and can be inserted explicitly in the code. This makes a frame pointer superfluous and leads to a significant simplification and speedup of prolog and epilog: Variables are addressed with SP as the base register.

Prolog	P	SUB	SP, SP, n+4	SP := SP-(n+4) (n = frame size)
		STW	LNK, SP, 0	push link
Epilog		LDW	LNK, SP, 0	pop link
		ADD	SP, SP, n+4	
		B	LNK	return jump

This is the scheme implemented in the Oberon-0 compiler. The handling of intermediate-level variables is not implemented.

Global variables have fixed addresses which must also be considered relative to a frame address. Their absolute values are determined upon loading the code, that is, *after* compilation but *before* program execution. The emitted object code can therefore be accompanied by a list of addresses of instructions referring to global variables. The loader must then add to these addresses the base address of the respective frame of global variables. This fixup operation can be omitted if the computer features the program counter as an address register. This is the case for example for the ARM architecture, but not in our RISC.

12.3. Parameters

Parameters constitute the interface between the calling and the called procedures. Parameters on the calling side are said to be *actual parameters*, and those on the called side *formal parameters*. The latter are in fact only place holders for which the actual parameters are substituted. Basically, a substitution is an assignment of the actual value to the formal variable.

This implies that every formal parameter be represented by a variable bound to the procedure, and that every call be accompanied by a number of assignments called parameter substitutions.

Most programming languages distinguish between at least two kinds of parameters. The first is the *value parameter* where, as its name suggests, the value of the actual parameter is assigned to the formal variable. The actual parameter is syntactically an expression. The second kind of parameter is the *reference parameter*, where, also as suggested by its name, a reference to the actual parameter is assigned to the formal variable. Evidently, the actual parameter must in this case be a variable, because an assignment to the formal parameter is permissible, and this assignment must refer to the actual variable. (In Pascal, Modula, and Oberon, the reference parameter is therefore called variable parameter). The value of the formal variable is in this case a hidden pointer, that is, an address.

Of course, the actual parameter must be evaluated before the substitution takes place. In the case of variable parameters, the evaluation takes the form of an identification of the variable, implying, for example, the evaluation of the index in the case of indexed variables. But how is the destination of this substitution to be determined? Here the stack organization of the store comes into play. The actual values are simply deposited in sequence on the top of the stack; no explicit destination addresses are required. Figure 12.4 shows the state of the stack after the deposition of the parameters, and after the call and the prologue.

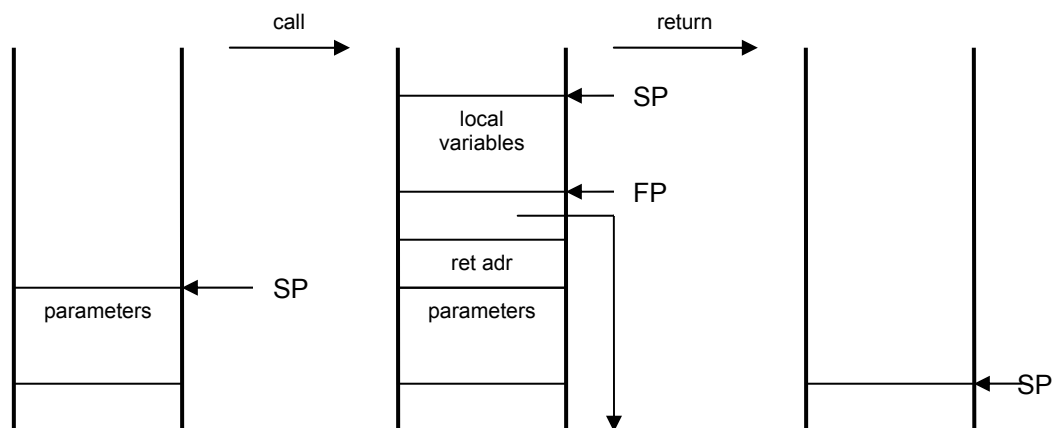


Figure 12.4. Parameter allocation.

It now becomes evident that parameters can be addressed relative to the frame address FP, like local variables. If local variables have negative offsets, parameters have positive offsets. It is particularly worth noting that the called procedure references parameters exactly where they were deposited by the calling procedure. The space allocated for the parameters is regained by the epilogue simply by carefully adjusting the value of SP.

In the case of computers with a large bank of registers, such as our RISC, a different scheme is available, namely to pass the parameters in these registers. The advantage is that code for depositing the parameters in memory exists only once, namely in the procedure's prolog. In the technique described earlier, code for storing occurs with each call. The drawback is that the number of parameters is limited (by the number of available registers). It is, however, poor programming practice to declare procedures with a large number of parameters. Thus this restriction is not one to be deplored. In memory, the parameters then simply precede the local variables.

12.4. Procedure declarations and calls

The procedure for processing procedure declarations is easily derived from the syntax with the aid of the parser construction rules. The new entry in the symbol table generated for a procedure declaration obtains the class attribute *Proc*, and its attribute *a* is given the current value of *pc*, which is the entry address of the procedure's prologue. Thereafter, a new scope is opened in the symbol table in such a way that (1) new entries for local objects are automatically inserted in the new scope, and (2) at the end of the procedure the local objects are easily discarded and the previous scope reappears. Here too, the two procedures *OpenScope* and *CloseScope* embody the stack principle, and the linkage is established by a header element (class *Head*, field *dsc*). Objects are given an additional attribute *lev* denoting the nesting level of the declared object. Consider the following declarations:

```
CONST N = 10;
VAR x: T;
PROCEDURE P(x, y: INTEGER); ...
```

The resulting symbol table is shown in Figure 12.5. The *dsc* pointer refers to *P*'s parameters *x* and *y*. Because of the two kind of parameters, it is necessary to introduce a new object-kind and item-mode. *Var* denotes variables and value parameters, the new *Par* denotes variable parameters. Value parameters are treated like local variables (class *Var*).

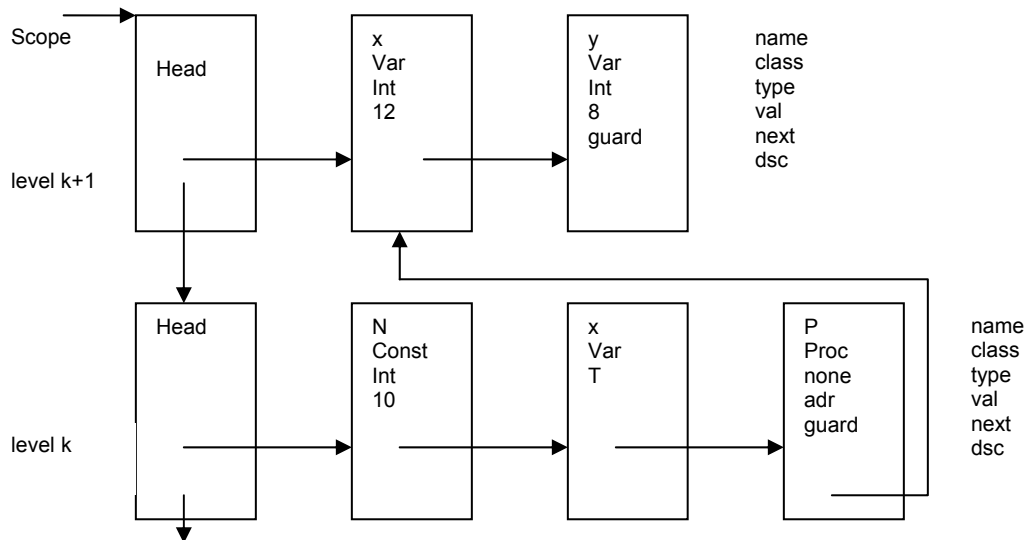


Figure 12.5. Symbol table representing two scopes.

```
PROCEDURE ProcedureDecl;
VAR proc, obj: Object;
    procid: Ident;
    locblksize, parblksize: LONGINT;

PROCEDURE FPSSection;
    VAR obj, first: Object; tp: Type; parsize: LONGINT;
BEGIN
    IF sym = var THEN Get(sym); IdentList(Par, first) ELSE IdentList(Var, first) END ;
    IF sym = ident THEN
        find(obj); Get(sym);
        IF obj.class = Typ THEN tp := obj.type ELSE Mark("type?"); tp := intType END
    ELSE Mark("ident?"); tp := intType
    END ;
    IF first.class = Var THEN parsize := tp.size ELSE parsize := 4 END ;
    obj := first;
    WHILE obj # guard DO
        obj.type := tp; obj.val := parblksize; obj.isparam := TRUE;
```

```

        INC(parblksize, parsize); obj := obj.next
    END
END
END FPSection;
BEGIN (* ProcedureDecl *) Get(sym);
IF sym = ident THEN
    procid := id;
    NewObj(proc, Proc); Get(sym); parblksize := 4;
    INC(level); OpenScope; proc.val := -1;
    IF sym = lparen THEN Get(sym);
        IF sym = rparen THEN Get(sym)
        ELSE FPSection;
            WHILE sym = semicolon DO Get(sym); FPSection END ;
            IF sym = rparen THEN Get(sym) ELSE Mark(")?") END
        END
    END ;
    locblksize := parblksize; proc.dsc := topScope.next;
    IF sym = semicolon THEN Get(sym) ELSE Mark(")?") END;
    declarations(locblksize);
    WHILE sym = procedure DO
        ProcedureDecl;
        IF sym = semicolon THEN Get(sym) ELSE Mark(")?") END
    END ;
    proc.val := pc; Enter(parblksize, locblksize);
    IF sym = begin THEN Get(sym); StatSequence END ;
    IF sym = end THEN Get(sym) ELSE Mark("END?") END ;
    IF sym = ident THEN
        IF procid # id THEN Mark("no match") END ;
        Get(sym)
    ELSE Mark("ident?")
    END ;
    Return(locblksize - 8); CloseScope; DEC(level)
END
END ProcedureDecl;

```

In the case of byte-addressed stores it is advantageous always to increment or decrement the stack pointer by multiples of 4, such that parameters are always aligned to word boundaries. In the case of Oberon-0 special attention to this rule is unnecessary, because all data types feature a size of multiples of 4 anyway.

Local declarations are processed by the parser procedure *declarations*. The code for the prologue is emitted by procedure *Enter* after the processing of local declarations. Emission of the epilogue is performed by procedure *Return* at the end of *ProcedureDecl*.

```

PROCEDURE Enter(parblksize, locblksize: LONGINT);
BEGIN a := 4; r := 0;
    Put1(SUB, SP, SP, locblksize); Put2(Stw, LNK, SP, 0);
    WHILE a < parblksize DO Put2(Stw, r, SP, a); INC(R); INC(a, 4) END
    (*store parameters from R0, R1, ... *)
END Enter;

PROCEDURE Return(size: LONGINT);
BEGIN
    Put2(Ldw, LNK, SP, 0); Put1(Add, SP, SP, size); Put3(0, 7, LNK)
END Return;

```

Procedure *Makeltem* converts a given object into a corresponding *Item*. At this point, the difference between the addressing of local and global variables must be taken into account. (As already mentioned, the handling of intermediate-level variables is not treated here.)

```

PROCEDURE Makeltem(VAR x: Item; y: Object);
    VAR r: LONGINT;

```



```

BEGIN x.mode := y.class; x.type := y.type; x.a := y.val;
  IF y.lev = 0 THEN x.r := SB ELSIF y.lev = curlev THEN x.r := SP
  ELSE Mark("level!"); x.r := 0
  END
END Makeltem;

```

Procedure calls are generated within the already encountered procedure *StatSequence* with the aid of auxiliary procedures *Parameter* and *Call*. The former calls on *load* (which is extended by the case for mode *Par*) and *loadAdr*, necessary for variable parameters. Similar addition are necessary in procedure *Store*.

```

IF sym = ident THEN
  find(obj); Get(sym); Makeltem(x, obj); selector(x);
  IF sym = becomes THEN ...

  ELSIF x.mode = Proc THEN
    par := obj.dsc;
    IF sym = lparen THEN Get(sym);
    IF sym = rpren THEN Get(sym)
    ELSE
      LOOP expression(y);
        IF par.isparam THEN
          IF y.type = par.type THEN Parameter(y, par.class)
          ELSE Mark("bad param type")
          END ;
          par := par.next
        ELSE Mark("too many parameters")
        END ;
        IF sym = comma THEN Get(sym)
        ELSIF sym = rpren THEN Get(sym); EXIT
        ELSIF sym >= semicolon THEN Mark(" ?"); EXIT
        ELSE Mark(" or , ?")
        END
      END
    END
  END ;
  IF obj.val < 0 THEN Mark("forward call")
  ELSE Call(x);
  IF par.isparam THEN Mark("too few parameters") END
  END
  ...
PROCEDURE Parameter(VAR x: Item; class: INTEGER);
  VAR r: LONGINT;
BEGIN
  IF class = Par THEN (*Var param*) loadAdr(x)
  ELSE (*value param*) load(x)
  END
END Parameter;

PROCEDURE Call(VAR x: Item);
BEGIN Put3(3, 7, x.a - pc-1); R := 0
END Call;

PROCEDURE load(VAR x: Item);
BEGIN
  IF x.mode # Reg THEN
    IF x.mode = Var THEN Put2(Ldw, R, x.r, x.a); INC(R)
    ELSIF x.mode = Par THEN Put2(Ldw, R, x.r, x.a); Put2(Ldw, R, R, 0); INC(R)
    ...
  END ;
  x.mode := Reg; x.r := R-1

```

```

END
END load;
PROCEDURE loadAdr(VAR x: Item);
BEGIN
  IF x.mode = Var THEN Put1(Add, R, x.r, x.a); INC(R)
  ELSIF x.mode = Par THEN Put2(Ldw, R, x.r, x.a); INC(R)
  ELSIF x.mode = Regl THEN Put2(Add, x.r, x.r, x.a)
  ELSE OSS.Mark("address error")
  END ;
  x.mode := Reg
END loadAdr;

PROCEDURE Store*(VAR x, y: Item); (* x := y *)
  VAR r: LONGINT;
BEGIN
  IF y.mode # Reg THEN load(y) END ;
  IF x.mode = Var THEN Put2(Stw, y.r, x.r, x.a); DEC(R)
  ELSIF x.mode = Par THEN Put2(Ldw, R, x.r, x.a); Put2(Stw, y.r, R, 0); DEC(R)
  ELSIF x.mode = Regl THEN Put2(Stw, y.r, x.r, x.a); DEC(R, 2)
  ELSE OSS.Mark("illegal assignment")
  END
END Store;

```

Here we tacitly assume that the entry addresses of procedures are known when a call is to be compiled. Thereby we exclude forward references which may, for example, arise in the case of mutual, recursive referencing. If this restriction is to be lifted, the locations of forward calls must be retained in order that the branch instructions may be fixed up when their destinations become known. This case is similar to the fixups required for forward jumps in conditional and repeated statements.

In conclusion, we show the code generated for the following, simple procedure:

```

PROCEDURE P(x: INTEGER; VAR y: INTEGER);
BEGIN x := y; y := x; P(x, y); P(y, x)
END P

```

0	SUB	SP, SP, 12	prolog
1	STW	LNK, SP, 0	
2	STW	R0, SP, 4	v
3	STW	R1, SP, 8	adr(y)
4	LDW	R0, SP, 8	
5	LDW	R0, R0, 0	y
6	STW	R0, SP, 4	x := y
7	LDW	R0, SP, 4	x
8	LDW	R1, SP, 8	
9	STW	R0, R1, 0	y := x
10	LDW	R0, SP, 4	x
11	LDW	R1, SP, 8	adr(y)
12	BL	-13	P(x, y)
13	LDW	R0, SP, 8	
14	LDW	R0, R0, 0	y
15	ADD	R1, SP, 4	adr(x)
16	BL	-17	P(y, x)
17	LDW	LNK, SP, 0	epilog
18	ADD	SP, SP, 12	
19	B	LNK	return

12.5. Standard procedures

Most programming languages feature certain procedures and functions which do not need to be declared in a program. They are said to be *predeclared* and they can be called from anywhere, as they are pervasive. These are well-known functions, type conversions, or frequently encountered statements which merit an abbreviation and are available on many computers as single instructions. The property common to all these so-called *standard* procedures is that they correspond directly either to a single instruction or to a short sequence of instructions. Therefore, these procedures are handled quite differently by compilers; no call is generated. Instead, the necessary instructions are emitted directly into the code. These procedures are therefore also called *in-line* procedures, a term that makes sense only if the underlying implementation technique is understood.

As a consequence it is advantageous to consider standard procedures as an object class of their own. Thereby the need for a special treatment of calls becomes immediately apparent. Here we consider only some predefined procedures particular to this compiler, which are used to generate output. Among standard functions ORD stands as a representative of all other function.

ORD(b)	INTEGER	the ordinal number of (Boolean) b
eot()	BOOLEAN	"the end of the input text has been reached"
ReadInt(n)		scan the input, assign integer to n
WriteInt(n)		output the integer n
WriteChar(n)		output the character, whose ordinal number is n
WriteLn		end the current line

The corresponding entries in the symbol table are made when the compiler is initialized, namely in an outermost scope called *universe* which always remains open. The new class attribute is denoted by *SProc*, and attribute *val* (*a* in the case of *Items*) identifies the concerned procedure. The following are the relevant excerpts from procedures *factor* and *StatSequence* respectively. The parameter *obj.val* specifies the number of the standard procedure or function. Further details can be extracted from the program listings.

```
IF sym = OSS.ident THEN
  find(obj); OSS.Get(sym);
  IF obj.class = OSG.SFunc THEN StandFunc(x, obj.val); x.type := obj.type
  ELSE OSG.MakeItem(x, obj); selector(x)
END
```

```
IF sym = ident THEN
  find(obj); Get(sym);
  IF obj.class = OSG.SProc THEN StandProc(obj.val)
  ELSE MakeItem(x, obj); selector(x);
  IF sym = becomes THEN ...
  ELSIF sym = lparen THEN ...
```

12.6. Function procedures

A function procedure is a procedure whose identifier simultaneously denotes both an algorithm and its result. It is activated not by a call statement but by a factor of an expression. The call of a function procedure must therefore also take care of returning the function's result. The question therefore arises of which resources should be used.

If our primary goal is the generation of efficient code with the minimal number of memory accesses, then a register is the prime candidate for temporarily holding the function's result. If this solution is adopted, we must renounce the capability of defining functions with a structured result, because structured values cannot be held in a register.

If this restriction is considered as unacceptable, a place in the stack must be reserved to hold the structured result. Typically, it is added to the parameter area of the activation record. The function result is considered as an implicit result (variable) parameter. Correspondingly, the stack pointer is adjusted before code for the first parameter is emitted.

At this point, all the concepts contained in the language Oberon-0 and implemented in its compiler have been presented.

12.7. Exercises.

- 12.1. Improve the Oberon-0 compiler in such a way that the restriction that variables must be strictly local or entirely global can be lifted.
- 12.2. Add standard functions to the Oberon-0 compiler, generating inline code. Consider ABS, INC, DEC.
- 12.3. Replace the VAR parameter concept by the notion of an OUT parameter. An OUT parameter represents a local variable whose value is assigned to its corresponding actual parameter upon termination of the procedure. It constitutes the inverse of the value parameter, where the value of the actual parameter is assigned to the formal variable upon the start of the procedure.

13. Elementary Data Types

13.1. The types REAL and LONGREAL

As early as 1957 integers and real numbers were treated as distinct data types in Fortran. This was not only because different, internal representations were necessary, but because it was recognized that the programmer must be aware of when computations could be expected to be exact (namely for integers), and when only approximate. The fact that with real numbers only approximate results can be obtained, may be understood by considering that real numbers are represented by scaled integers with a fixed, finite number of digits. Their type is called REAL, and a real value x is represented by the pair of integers e and m as defined by the equation

$$x = B^{e-w} \times m \quad 1 \leq m < B$$

This form is called *floating-point* representation; e is said to be the exponent, m the mantissa. The base B and the bias w are fixed values for all REAL values, characterizing the chosen number representation. The two IEEE standards of floating-point representations feature the following values for B and w , and to the components e and m a bit s is added for the sign:

Type	B	w	Number of bits for e	Number of bits for m	Total
REAL	2	127	8	23	32
LONGREAL	2	1023	11	52	64

The exact forms of the two types, called REAL and LONGREAL in Oberon, are specified by the following formulas:

$$x = (-1)^s \times 2^{e-127} \times 1.m \quad x = (-1)^s \times 2^{e-1023} \times 1.m$$

The following examples show the floating-point representation of some selected numbers:

Decimal	s	e	1.m	Binary	Hexadecimal
1.0	0	127	1.0	0 01111111 000000000000000000000000	3F80 0000
0.5	0	126	1.0	0 01111110 000000000000000000000000	3F00 0000
2.0	0	128	1.0	0 10000000 000000000000000000000000	4000 0000
10.0	0	130	1.25	0 10000010 010000000000000000000000	4120 0000
0.1	0	123	1.6	0 01111011 10011001100110011001101	3DC CCCD
-1.5	1	127	1.5	1 01111111 100000000000000000000000	BFC0 0000

Two examples illustrate the case of LONGREAL:

1.0	0	1023	1.0	0 011111111111 00000000 ... 00000000	3FF0 0000 0000 0000
0.1	0	1019	1.6	0 011111110111 10011001 ... 10011010	3FB9 9999 9999 999A

This logarithmic form inherently excludes a value for 0. The value 0 must be treated as a special case, and it is represented by all bits being 0. With regard to numeric properties it constitutes a special case and a discontinuity. Furthermore, the IEEE standards postulate two additional special values: $e = 0$ (with $m \neq 0$) and $e = 255$ (resp. $e = 1023$) are considered as invalid results and they are called *NaN* (not a number).

Normally, the programmer does not have to worry about these specifications, and the compiler designer is not affected by them. The types REAL and LONGREAL constitute abstract data types usually integrated in the hardware which features a set of instructions adapted to the floating-point representation. If this set is complete, that is, it covers all basic numeric operations, the representation may be considered as hidden, since no further, programmed operations depend on it. In many computers, instructions for floating-point operands use a special set of registers. The reason behind this is that often separate coprocessors, so-called *floating-point units* (FPUs) are used which implement all floating-point instructions and contain this set of floating-point registers.

13.2. Compatibility between numeric data types

The values of all variables with numeric data type are numbers. Therefore there is no obvious reason not to declare them all as assignment compatible. But, as already outlined, numbers of different types are differently represented in terms of bit sequences within the computer. Hence, whenever a number of type T_0 is assigned to a variable of type T_1 , a representation conversion has to be performed which takes little time. The question then arises of whether this fact should remain hidden from the programmer in order to avoid distraction, or whether it should be made explicit because it affects the efficiency of the program. The latter choice is accomplished by declaring the various types as incompatible and by providing explicit, predefined conversion functions.

In any case, in order to be complete, a computer's set of instructions must also contain conversion instructions which convert integers into floating-point numbers and vice-versa. The same holds at the level of the programming language.

In (revised) Oberon there are only the arithmetic data types: INTEGER, REAL, BYTE, and possibly LONGREAL. There are two conversion functions:

FLT: INTEGER \rightarrow REAL

FLOOR: REAL \rightarrow INTEGER

FLOOR(x) yields the largest integer less or equal to x. For example

FLOOR(1.5) = 1 FLOOR(-1.5) = -2

The type BYTE is a *subrange* of INTEGER with values 0 .. 255, and therefore no conversion functions are needed. However, assigning to a variable of type BYTE may cause overflow. Typically, arithmetic operations are performed on values of type INTEGER, that is, the compiler treats variables of type BYTE with form *Int* (see Ch. 8).

13.3. The data type SET

The units of storage in computers consist of a small number of bits which are interpretable in different ways. They may represent integers with or without sign, floating-point numbers or logical data. The question about the way to introduce logical bit sequences in higher programming languages has been controversial for a long time. The proposal to introduce them as sets is due to C. A. R. Hoare (Hoare, 1972).

The proposal is attractive, because the set is a mathematically well-founded abstraction. It is appropriately represented in a computer by its *characteristic function* F . If x is a set of elements from the ordered base set M , $F(x)$ is the sequence of truth values b_i with the meaning " i is contained in x ". If we choose a word (consisting of N bits) to represent values of type SET, the base set consists of the integers 0, 1, ..., $N-1$. N is typically so small that the range of applications for the type SET is quite restricted. However, the basic set operations of intersection, union and difference are implementable extremely efficiently. Examples of sets represented by bit sequences with word length 4 are:

<u>x</u>	<u>3 2 1 0</u>
{0, 2}	0 1 0 1
{0, 1, 3}	1 0 1 1
{ }	0 0 0 0

Oberon's set operators are implemented by logical instructions available on every computer. Note that we use the Oberon notation for set operations, that is, $x+y$ for the union, $x*y$ for the intersection, and $x-y$ for the difference. Consequently, the IOR instruction can be used for set union, AND for set intersection, ANN for the difference, and XOR for the symmetric difference.

The result is a very efficient implementation, because the operation is executed on all elements (bits) simultaneously (in parallel). Examples with the base set {0, 1, 2, 3} are:

$\{0, 1\} + \{0, 2\} = \{0, 1, 2\}$	0011 OR	0101 =	0111
$\{0, 1\} * \{0, 2\} = \{0\}$	0011 &	0101 =	0001
$\{0, 1\} - \{0, 2\} = \{1\}$	0011 & ~	0101 =	0010
$\{0, 1\} / \{0, 2\} = \{1, 2\}$	0011 XOR	0101 =	0110

We conclude by showing the code representing the set expression $(a+b) * (c+d)$

```
LDW  R0, base, a
LDW  R1, base, b
IOR  R0, R0, R1
LDW  R1, base, c
LDW  R2, base, d
IOR  R1, R1, R2
AND  R0, R0, R1
```

The membership test $i \text{ IN } sx$ is implemented by a bit test, typically a shift followed by a sign bit test..

```
LDW  R0, base, s
ROR  R0, R0, i+1
```

The type SET is particularly useful if the base set includes the ordinal numbers of a character set (CHAR). Efficiency is in this case somewhat reduced, because 256 bits (32 bytes) are typically required to represent a set value. Even in 32-bit computers 8 logical instructions are required for the execution of a set operation.

13.4. Exercises

13.1 Extend the language Oberon-0 and its compiler by the data type REAL (and/or LONGREAL) with its arithmetic operators +, -, * and /. The RISC architecture must be extended accordingly by a set of floating-point instructions. Choose one of the following alternatives:

a. The result type of an operation is always that of the operands. The types INTEGER and REAL cannot be mixed. However, there exist the two transfer functions $FLOOR(x)$ and $FLT(i)$.

b. Operands of the types INTEGER and REAL (and LONGREAL) may be mixed in expressions.

Compare the complexities of the compilers in the two cases.

13.2. Extend the language Oberon-0 and its compiler by the data type SET with its operators + (union), * (intersection) and - (difference), and with the relation IN (membership). Furthermore, set constructors are introduced by the following additional syntax. As an option, expressions in set constructors may be confined to constants.

```
factor = number | set | ...
set = "{" [element {"," element}] "}".
element = expression [".." expression].
```

13.3. Extend the language Oberon-0 and its compiler by the data type CHAR with the functions $ORD(ch)$ (ordinal number of ch in the character set) and $CHR(k)$ (k -th character in the character set). A variable of type CHAR occupies a single byte in store.

14. Open Arrays, Pointers and Procedure Types

14.1. Open arrays

An *open array* is an array parameter whose length is unknown (open) at the time of compilation. Here we encounter for the first time a situation where the size of the required storage block is not given. The solution is relatively simple in the case of a reference parameter, because no storage has to be allocated anyway, and merely a reference to the actual array is passed to the called procedure.

However, in order to check index bounds when accessing elements of the open array parameter, the length must be known. Therefore, in addition to the array's address, its length is also passed on. In the case of a multidimensional, open array the length is also necessary to compute element addresses. Hence, the length of the array in every dimension is supplied. The unit consisting of array address and lengths is called an *array descriptor*. Consider the following example:

```
VAR a: ARRAY 10 OF ARRAY 20 OF INTEGER;

PROCEDURE P(VAR x: ARRAY OF ARRAY OF INTEGER);
BEGIN k := x[i]
END P;

P(a)
```

Under the assumption that parameters are passed in registers, as shown in Ch. 12, the corresponding code is as follows:

```
MOV   R0, 20           R0 := 20
MOV   R1, 10           R1 := 10
ADD   R2, base, a     R2 := adr(a)
BL    P               call
```

If an open array parameter is passed by value, its value must be copied into its provided formal location just as in the case of a scalar value. This operation may, however, take considerable effort if the array is large. In the case of structured parameters, programmers should always use the VAR option, unless a copy is essential.

Certainly the code for the copy operation is better inserted after the prologue of the procedure rather than in the place of the call. Consequently, the code pattern for the call is the same for value and reference parameters, with the exception that for the former the copy operation is omitted from the prologue.

The formal location apparently does not hold the array, but instead the array descriptor, whose size is known. The space for the copy is allocated at the top of the stack, and the stack pointer is incremented (or decremented) by the array's size. In the case of multidimensional arrays, the size is computed (at run-time) as the product of the individual lengths and the element size.

Here SP is changed at run time by amounts which are unknown at compile time. Therefore it is impossible in the general case to operate with a single address register (SP); the frame pointer FP is indeed necessary.

14.2. Dynamic data structures and pointers

The two forms of data structures provided in Oberon are the array (all elements of the same type, *homogeneous* structure) and the record (*heterogeneous* structure). More complex structures must be programmed individually, that is, they must be generated during program execution. For this reason they are said to be *dynamic* structures. Thereby the structure's components are generated one by one; storage is allocated for components individually. They do not necessarily lie in contiguous locations in store. Relationships between components are expressed explicitly by *pointers*.

For the implementation of this concept a mechanism must be available for the allocation of storage at run time. In Oberon, it is represented by the standard procedure NEW(x). This allocates storage to a dynamic variable, and assigns the address of the allocated block to the pointer variable x. From this it follows that pointers are addresses. Access to a variable referenced by a pointer is necessarily indirect as in the case of VAR parameters. In fact, a VAR parameter represents a hidden pointer. Consider the following declarations:

```
TYPE T = POINTER TO TDesc;
      TDesc = RECORD x, y : LONGINT END;
VAR a, b : T;
```

The code for the assignment $a.x := b.y$ with access via pointers a and b becomes

```
LDW  R0, SP, b      R0 := b
LDW  R0, R0, y      R0 := b.y
LDW  R1, SP, a      R1 := a
STW  R0, R1, x      a.x := R0
```

The step from the referencing pointer variable to the referenced record variable is called *dereferencing*. In Oberon the explicit dereferencing operator is denoted by the symbol $\hat{\cdot}$. $a.x$ is evidently an abbreviation for the more explicit form $a\hat{\cdot}.x$. The implicit dereferencing operation is recognizable when the selector symbol (dot) is preceded not by a record but by a pointer variable.

Everyone who has written programs which heavily involve pointer handling knows how easily errors can be made with catastrophic consequences. To explain why, consider the following type declarations:

```
T0 = RECORD x, y : LONGINT END ;
T1 = RECORD x, y, z : LONGINT END;
```

Let a and b be pointer variables, and let a point to a record of type $T0$, b to a record of type $T1$. Then the designator $a.z$ denotes an undefined value of a non-existent variable, and $a.z := b.x$ stores a value to some undefined location, perhaps corrupting another variable allocated to this location.

This dangerous situation is elegantly eliminated by binding pointers to a data type. This permits the validation of pointer values at the time of compilation without loss of run-time efficiency. This brilliant idea is due to C. A. R. Hoare and was implemented for the first time in Algol W (Hoare, 1972). The type to which a pointer is bound is called its *base type*.

```
P0 = POINTER TO T0;
P1 = POINTER TO T1;
```

Now the compiler can check and guarantee that only pointer values can be assigned to a pointer variable p which points to a variable of the base type of p . The value NIL, pointing to no variable at all, is considered as belonging to all pointer types. Referring to the example above, now the designator $a.z$ is detected as incorrect, because z is not a field of the type $T0$ to which a is bound. If every pointer variable is initialized to NIL, it suffices to precede every access via a pointer with a test for the pointer value NIL. In this case, the pointer points to no variable, and any designator must be erroneous.

Such a test is indeed quite simple, but because of its frequency it reduces efficiency. The need for an explicit code pattern can be circumvented by (ab)using the storage protection mechanism available on many computers. In this case, the test does not properly check whether $a = \text{NIL}$, but rather whether $a.z$ is a valid, unprotected address. If as usual NIL is represented by the address 0, and if locations $0 \dots N-1$ are protected, mistaken references via NIL are caught only if their field offsets are less than N . Nevertheless, the method seems to be satisfactory in practice.

The introduction of pointers requires a new class of objects in the symbol table and also a new mode of items. Both are to imply indirect addressing. Because VAR parameters also require indirect addressing, a mode indicating indirection is already present, and it is only natural to use

the same mode for access via pointers. However, the name *Ind* would now appear as more appropriate than *Par*.

<u>Designator</u>	<u>Mode</u>	
x	Var	Direct addressing
x↑	Ind	Indirect addressing
x↑.y	Ind	Indirect addressing with offset

Hence, the (usually implied) dereferencing operator converts the mode of an item from *Var* to *Ind*. To summarize:

1. The notion of a pointer is easily integrated into our system of type compatibility checking. Every pointer type is bound to a base type, namely the type of the referenced variable.
2. x↑ denotes dereferencing, implemented by indirect addressing.
3. Pointers are type safe if access is preceded by a NIL test, and if pointer variables are initialized to NIL.

Allocation of variables referenced via pointers is obtained by a call of procedure *NEW(p)*. We postulate its existence as run-time support in operating systems. The size of the block to be allocated is given by the base type of *p*.

So far, we have ignored the problem of storage reclamation. It is actually irrelevant for abstract programs; for concrete ones, however, it is crucial, as stores are inherently finite. Modern operating systems offer a centralized storage management with garbage collection. There are various schemes for storage reclamation; but we shall not explain them here. We restrict ourselves to the only question relevant to the compiler designer: which data must be provided to the garbage collector, so that at any time all irrelevant storage blocks can safely be identified and reclaimed? A variable is no longer relevant when there are no references to it, references emanating from declared pointer variables. In order to determine whether such references exist, the garbage collector requires the following data:

1. the addresses of all declared pointer variables,
2. the offsets of all pointer fields in dynamically allocated records, and
3. the size of every dynamically allocated variable.

This information is available at compile time, and it has to be "handed down" in such a way that it is available to the garbage collector at run time. In this sense compiler and system must be integrated. The system is here assumed to include storage management, in particular the allocator *NEW* and the garbage collector.

In order to make this information available at run time, procedure *NEW* not only allocates a block of storage, but provides it with a type description of the allocated variable. Naturally, such a descriptor must be issued only once, as it need not be duplicated for every instance (variable) of the same type. Therefore, the block is assigned merely a pointer to the type descriptor, and this pointer remains invisible to the programmer. The pointer is called a *type tag* (s. Figure 14.1).

The type descriptor apparently is a reduced form of the object describing the type in the compiler's symbol table, reduced to the data relevant for storage reclamation. This concept has the following consequences:

1. The compiler must generate a descriptor for every (record) type, and it must add it to the object file.
2. Procedure *NEW(p)* obtains, in addition to the address of *p*, an additional, hidden parameter specifying the address of the descriptor of the base type of *p*.
3. The program loader must interpret the added object file information and generate type descriptors.

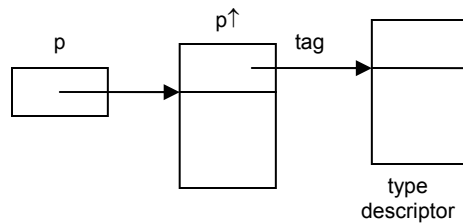


Figure 14.1. Pointer variable, referenced variable, and type descriptor.

The type descriptor specifies the size of the variable and the offset of all pointer fields (Figure 14.2).

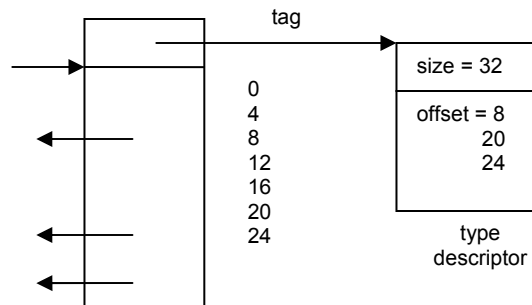


Figure 14.2. Variable with type descriptor.

This, however, is still insufficient. In order that data structures can be traversed, their roots have to be known. Therefore, the object file is also provided with a list of all declared pointer variables. This list is copied upon loading into memory. The list must also include the hidden pointers designating type descriptors. In order that descriptors do not have to be generated for all data types, Oberon restricts pointers to refer to records. This is justified when considering the role of records in dynamic data structures.

14.3. Procedure types

If in a language procedures can be passed as parameters, or if they can occur as values of variables, it becomes necessary to introduce *procedure types*. Which are the characteristics of such types, that is, of the values which variables may assume?

Procedure types have been in use since the advent of Algol 60. There, they occurred implicitly only. A parameter in Algol 60 can be a procedure (formal procedure). Its type, however, is not specified; it is merely known that the parameter denotes some procedure or function. The type specification is incomplete or missing, and this constitutes an unfortunate loophole in Algol's type system. In Pascal, it was retained as a concession to Algol compatibility. Modula-2, however, requires a complete, type-safe specification, and besides parameters, variables with procedures as their values are also allowed. Thereby procedure types achieve the same standing as other data types. In this respect, Oberon has adopted the same concept as Modula-2 (Wirth, 1982).

What does this type-safe specification, called the procedure's *signature*, consist of? It contains all specifications necessary to validate the compatibility between actual and formal parameters, namely their number, the type of each parameter, its kind (value or reference) and the type of the result in the case of function procedures. The following example illustrates the case:

```
PROCEDURE F(x, y : REAL): REAL;
BEGIN
```

```

...
END F
PROCEDURE H(f: PROCEDURE (u, v : REAL): REAL);

BEGIN a := f(a + b, a - b)
END H

```

VAR a, b: REAL;

Upon compilation of the declaration of H the type compatibility between $a + b$ and u , respectively that between $a - b$ and v , is checked, as well as whether the result type of f is assignable to a . In the call $H(F)$ the compatibility between the parameters, and that of the result type of the actual F and the formal f is verified, that is, between x and u and between y and v . Note that the identifiers u and v do not occur in the program, except as the names of the formal parameters of the formal procedure f . Hence, they are actually superfluous, but they may be useful as comments to the reader if meaningful names are chosen.

Pascal, Modula and Oberon assume *name compatibility* as the basis for establishing type consistency. In the case of procedure parameters, an exception was made; *structural compatibility* suffices. If name compatibility were required, the type (signature) of every procedure used as an actual parameter would have to be given an explicit name. This was considered as too cumbersome when the language was designed. However, structural compatibility requires that a compiler be capable of comparing two parameter lists for type correspondence.

A procedure may thus be assigned to a variable under the condition that the two parameter lists correspond. The assigned procedure is activated by referring to the procedure variable. The call is indirect. This is actually the basis of object-oriented programming, where procedures are bound to fields of record variables called *objects*. These bound procedures are called methods. In contrast to Oberon, methods, once declared and bound, cannot be altered. All instances of a class refer to the same methods.

The implementation of procedure types and methods turns out to be surprisingly simple, if the problem of type compatibility checking is ignored. The value of a variable or record field with procedure type is simply the entry address of the assigned procedure. This holds only if we require that only global procedures, that is, procedures which are not embedded in some context, can be assigned. This readily acceptable restriction is explained with the aid of the following example which breaches this restriction. Upon execution of Q alias v the context containing variables a and b is missing.

```

TYPE T = PROCEDURE (u: INTEGER);
VAR v: T; r: INTEGER;

PROCEDURE P;
  VAR a, b: INTEGER;
  PROCEDURE Q(VAR x: INTEGER);
    BEGIN x := a+b END Q;
  BEGIN v := Q
  END P;

... P; v(r) ...

```

14.4. Exercises.

14.1. Extend the language Oberon-0 and its compiler with open arrays:

- a. for one-dimensional VAR parameters,
- b. for multi-dimensional VAR parameters.

14.2. Extend the language Oberon-0 and its compiler with function procedures:

- a. for scalar result types (INTEGER, REAL, SET),
- b. for any type.

14.3. A certain module M manages a data structure whose details are to be kept hidden. In spite of this hiding it must be possible to apply any given operation P on all elements of the structure. For this purpose, a procedure *Enumerate* is exported from M , which allows P to be specified as parameter. As a simple example, we choose for P the counting of the elements currently in the data structure and display the desired solution:

```
PROCEDURE Enumerate(P: PROCEDURE (e: Element));  
PROCEDURE CountElements*;  
  VAR n: INTEGER;  
  PROCEDURE Cnt(e: Element); BEGIN n := n+1 END Cnt;  
BEGIN n := 0; M.Enumerate(Cnt); Texts.WriteInt(W, n, 6)  
END CountElements;
```

Unfortunately, this solution violates a restriction postulated for the language Oberon. The restriction specifies that procedures used as parameters must be declared globally. This forces us to declare *Cnt* outside of *CountElements* and thereby also the counter variable n , although both definitely have no global function.

Implement procedure types in such a way that the mentioned restriction can be lifted, and that the proposed solution is admissible. What is the price?

15. Modules and Separate Compilation

15.1. The principle of information hiding

Algol 60 introduced the principles of textual locality of identifiers and that of limited lifetime of the identified objects during execution. The range of visibility of an identifier in the text is called scope (Naur, 1960), and it extends over the block in which the identifier is declared. According to the syntax, blocks may be nested, with the consequence that the rules about visibility and scopes must be refined. Algol 60 postulates that identifiers declared in a block B are visible within B and within all blocks contained in B. But they are invisible in the environment of B.

From this rule, the implementer concludes that storage must be allocated to a variable *x* local to B as soon as control enters B, and that storage may be released as soon as control leaves B. Not only is *x* invisible outside B, but *x* ceases to exist when control is outside B. This implies the significant advantage that storage need not remain allocated to all variables of a program.

In some cases, however, the continued existence of a variable during a period of invisibility is highly desirable. Variable *x* then seems to reappear with its previous value as soon as control enters block B again. This special case was covered in Algol 60 by the feature of own variables. But this solution was soon discovered to be quite unsatisfactory, in particular in connection with recursive procedures.

An elegant and highly useful solution to the own-problem was discovered around 1972 with the structure of the module. It was adopted in the languages Modula (Wirth, 1977) and Mesa (Mitchell, Maybury and Sweet, 1978), and later under the name package in Ada. Syntactically, a module resembles a procedure and consists of local declarations followed by statements. In contrast to a procedure, however, a module is not called, but its statements are executed once only, namely when the module is loaded. The locally declared objects are static and remain in existence as long as the module remains loaded. The statements in the module body merely serve to initialize the module's variables. These variables are invisible outside the module; effectively they are hidden. D. L. Parnas has coined the term information hiding, and it has become an important notion in software construction. Oberon features the possibility of specifying selected identifiers declared in modules as visible in the module's environment. These identifiers are then said to be exported.

The own variable *x* declared within the Algol procedure P now will be declared, like P itself, local to a module M. P is exported, but not *x*. In the environment of M the details of the implementation of P as well as the variable *x* are hidden, but *x* retains its existence and its value between calls of P.

The desire to hide certain objects and details is particularly justified if a system consist of various parts whose tasks are relatively well separated, and if the parts themselves are of a certain complexity. This is typically the case in an organizational unit which manages a data structure. Then the data structure is hidden within a module, and it is accessible only via exported procedures. The programmer of this module postulates certain invariants, such as consistency conditions, which govern the data structure. These invariants can be guaranteed to hold, because they cannot be violated by parts of the system outside the module. As a consequence, the programmer's responsibility is effectively limited to the procedures within the module. This encapsulation of details solely responsible for the specified invariants is the true purpose of information hiding and of the module concept.

Typical examples of modules and information hiding are the file system hiding the structure of files and their dictionary, the scanner of a compiler hiding the source text and its lexicographic structure, or the code generator of a compiler hiding the generated code and the structure of the target architecture.

15.2. Separate compilation

It is tempting to postulate that modules be nestable like procedures. This facility is offered for example by the language Modula-2. In practice, however, this flexibility has hardly been fruitful. A flat module structure usually suffices. Hence, we consider all modules as being global, and their environment as the universe.

Much more relevant than their nestability is the possibility of developing and compiling modules separately. The latter is clearly feasible only if the modules are global, that is, not nested. The reason for this demand is simply the fact that software is never planned, implemented and tested in straight sequence, but that it is developed in steps, each step incorporating some additions or adaptations. Software is not "written", but grows. The module concept is of fundamental importance in this connection, because it allows development of individual modules separately under the assumption of constant interfaces of their imports. The set of exported objects effectively constitutes a module's interface with its partners. If an interface remains unchanged, a module's implementation can be improved (and corrected) without needing to adapt and recompile the module's clients. This is the real justification for separate compilation.

The advantage of this concept becomes particularly relevant if software is developed by teams. Once agreement is reached about the partitioning of a system into modules and about their interfaces, the team members can proceed independently in implementing the module assigned to them. Even if in practice it turns out that later changes in the specification of interfaces are avoidable only rarely, the simplification of teamwork through the concept of separate compilation of modules can hardly be overestimated. The successful development of complex systems crucially depends on the concept of modules and their separate compilation.

At this point, the reader may think that all this is not new, that the independent programming of modules and their binding by the program loader, as symbolized in Figure 15.1, has been in common use since the era of assemblers and the first Fortran compilers.

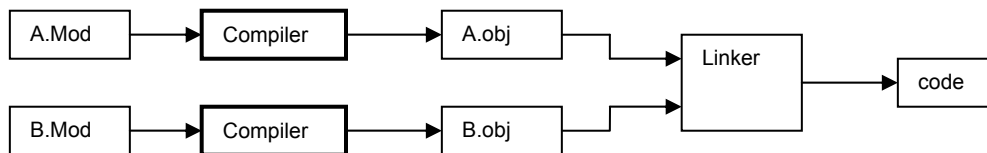


Figure 15.1. Independent compilation of modules A and B.

However, this ignores the fact that higher programming languages offer significantly increased protection against mistakes and inconsistencies through their static type concept. This inestimable - but all too often underestimated gain - is swept aside if type consistency checks are guaranteed only within modules, but not across module boundaries. This implies that type information about all imported objects must be available whenever a module is compiled. In contrast to independent compilation (Figure 15.1), where this information is not available, compilation as shown in Figure 15.2 with type consistency checks across module boundaries is called separate compilation.

Information about the imported objects is essentially an excerpt of the symbol table as presented in Chapter 8. This excerpt of the symbol table, transformed into a sequential form, is called a symbol file. Compilation of a module A which imports (objects from) modules B1 ... Bn now requires, in addition to the source text of A, the symbol files of B1 ... Bn. And in addition to the object code (A.obj) it also generates a symbol file (A.sym).

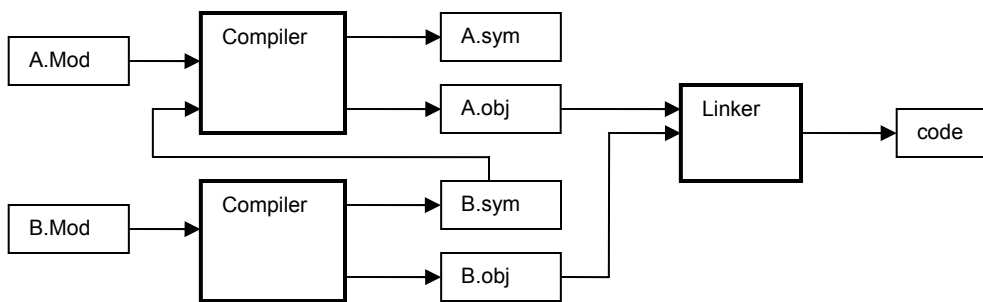


Figure 15.2. Separate compilation of modules A and B.

15.3. Implementation of symbol files

From the foregoing considerations we may first conclude that compilation of a module's import list causes a symbol file to be read for each module identifier in the list. The symbol table of the compiled module is initialized by the imported symbol files. Second, it follows that at the end of compilation the new symbol table is traversed, and a symbol file is output with an entry corresponding to every symbol table element marked for export. Figure 15.3 shows as an example the relevant excerpt of the symbol table during compilation of a module A importing B. Within B, T and f are marked with an asterisk for export.

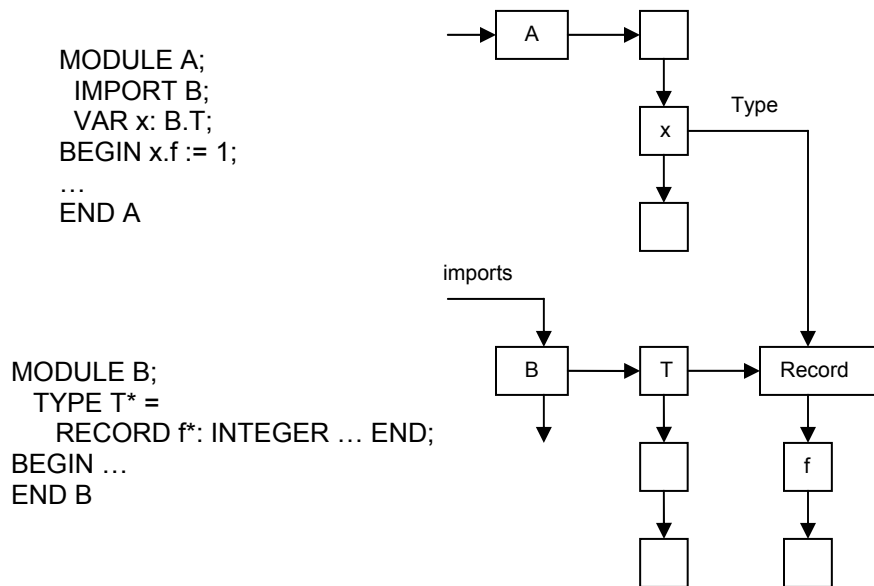


Figure 15.3. Symbol table of A with imports from B.

Let us first consider the generation of the symbol file M.sym of a module M. At first sight, the task merely consists of traversing the table and emitting an entry corresponding to every marked element in an appropriately sequentialized form. The symbol table is essentially a list of objects with pointers to type structures which are trees. In this case the sequentialization of structures using a characteristic prefix for every element is perhaps the most appropriate technique. It is illustrated by an example in Figure 15.4.

```

VAR x: ARRAY 10 OF INTEGER;
    y: ARRAY 8 OF ARRAY 20 OF REAL
  
```

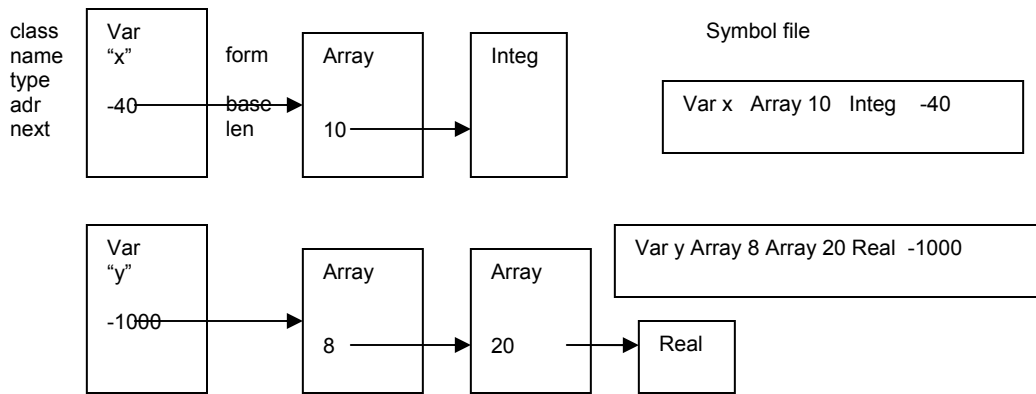



Figure 15.4. Sequentialized form of a symbol table with two arrays.

A problem arises because every object contains at least a pointer referring to its type. Writing pointer values into a file is problematic, to say the least. Our solution consists in writing the type description into the file the first time it is encountered when scanning the symbol table. Thereby the type entry is marked and obtains a unique reference number. The number is stored in an additional record field of the type ObjectDesc. If the type is referenced again later, the reference number is output instead of the structure.

This technique not only avoids the repeated writing of the same type descriptions, but also solves the problem of recursive references, as shown in Figure 15.5.

TYPE P = POINTER TO R;
R = RECORD x, y: INTEGER; next: P END

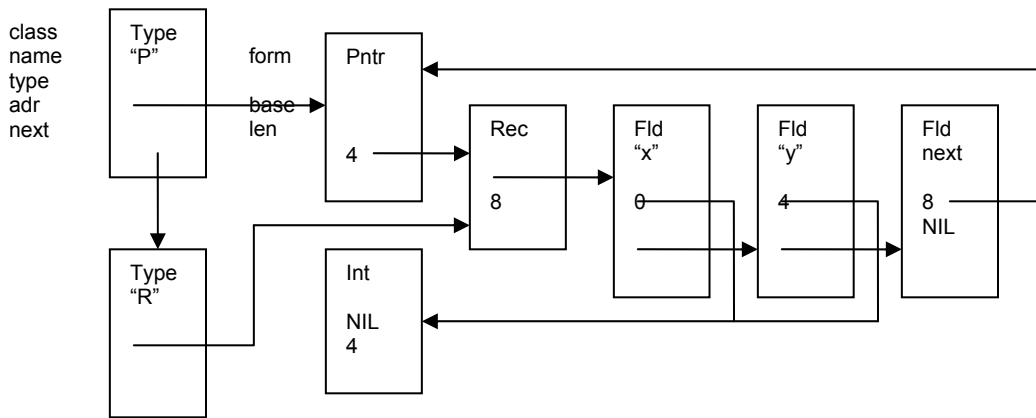


Figure 15.5. Cyclic reference of type node.

Positive values are used for reference numbers. As an indication that the reference number is used for the first time, and that it is therefore immediately followed by the type description, the number is given a negative sign. While reading a symbol file, a type table T is constructed with references to the respective type structures. If a positive reference number r is read, T[r] is the needed pointer; if r is negative, the subsequent type data is read, and the pointer referring to the newly constructed descriptor is assigned to T[-r].

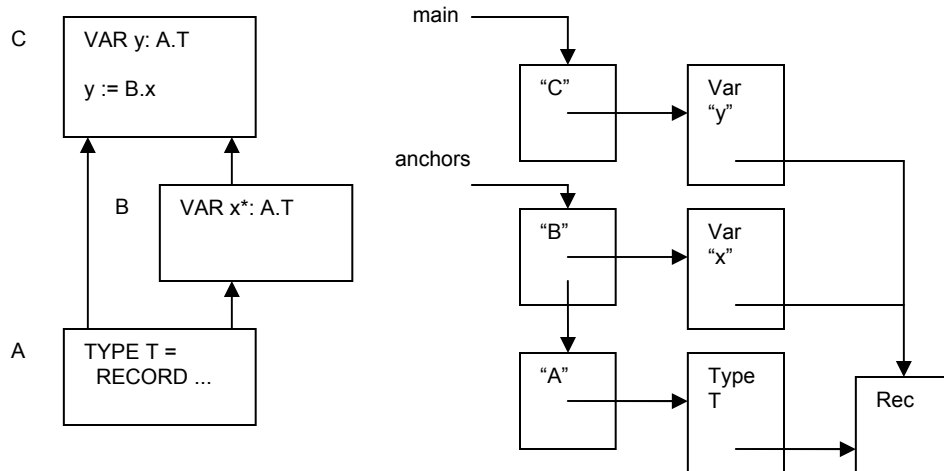


Figure 15.6. Re_export of type A.T from module B.

Type information can, in contrast to data about other objects, be imported and at the same time be re-exported. Therefore it is necessary to specify the module from which the exported type stems. In order to make this possible, we use a so-called module anchor. In the heading of every symbol file there is a list of anchor objects, one for each imported module which is re_exported, that is, which contains a type that is referenced by an exported object. Figure 15.6 illustrates such a situation; module C imports modules A and B, whereby a variable x is imported from B whose type stems from A. The type compatibility check for an assignment like $y := x$ rests on the assumption that the type pointers of x and y both refer to the same type descriptor. If they do not, an error is indicated.

Hence we conclude that upon compilation of a module M, not only the symbol tables of the explicitly imported modules must be present, but also those of modules from which types are referenced either directly or indirectly. This is a cause for concern, because the compilation of any module might necessitate the reading of symbol files of entire module hierarchies. It might even reach down to the deepest level of an operating environment, from where neither variables nor procedures are imported, but perhaps only a single type. The result would not only be the superfluous loading of large amounts of data, but also a waste of much memory space. It turns out, that although our concern is justified, the consequences are much less dramatic than might be expected (Franz, 1993). The reason is that most symbol tables requested are present already for other reasons. As a consequence, the additional effort remains small. Nevertheless it is worth pondering over the possibility of avoiding the extra effort. Indeed, the first compilers for Modula and Oberon have adopted the following technique.

Let a module M import types from modules M0, M1, and so on, either directly or indirectly. The solution consists of including in the symbol file of M complete descriptions of the imported types, thereby avoiding references to the secondary modules M0, M1, and so on. However, this fairly obvious solution causes complication. In the example illustrated by Figure 15.6, the symbol file of B evidently contains a complete description of type T. The consistency check for the assignment $y := B.x$, in order to be highly efficient, merely compares two type pointers. The configuration shown on the right of Figure 15.6 must therefore be present after loading. This implies that in symbol files re-exported types not only specify their home module, but that when loading a symbol file a test must verify whether or not the read type is already present. This may be the case because the symbol file of the module defining the type has already been loaded, or because the type has already been read when loading other symbol files.

At this point we also mention another, small complication in connection with types that arises because types may appear under different names (aliases). Although use of aliases is rare, the

language definition (unfortunately) allows it. They are moderately meaningful only if the synonyms stem from different modules, as shown in Figure 15.7.

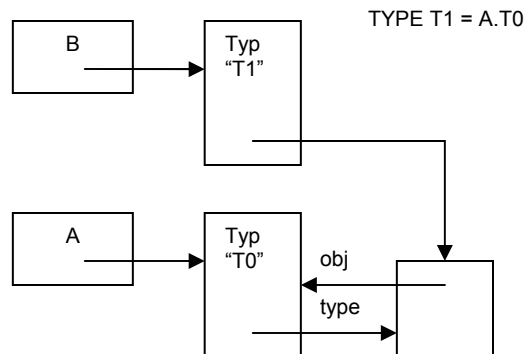


Figure 15.7. Type with aliases.

When loading the symbol file of B it is recognized that B.T1 and A.T0, both pointing to a type object, must actually point to the same object descriptor. In order to determine which of the two descriptors should be discarded and which one retained, type nodes (type Structure) are supplied with a back-pointer to the original type object (type Object), here to T0.

15.4. Addressing external objects

The principal advantage of separate compilation is that changes in a module M do not invalidate clients of M, if the interface of M remains unaffected. We recall that the interface consists of the entire set of exported declarations. Changes which do not affect the interface may occur, so to say, under cover, and without client programmers being aware of them. Such changes must not even require recompilation of the clients using new symbol files. For the sake of honesty, we hasten to add that exported procedures must in their semantics not have altered, because compilers could not detect such changes reliably. Hence, if we say that an interface remains unchanged, we explicitly refer to the declarations of types and variables, and to the signatures of procedures, and only implicitly to their semantics.

If in a certain module non-exported procedures and variables are changed, added or deleted, their addresses necessarily also change, and as a consequence so do those of other, possibly exported variables and procedures. This leads to a change of the symbol table, and thereby also to an invalidation of client modules. But this obviously contradicts the requirements postulated for separate compilation.

The solution to this dilemma lies in avoiding the inclusion of addresses in a symbol file. This has the consequence that addresses must be computed when loading and binding a module. Hence, in addition to its address (for module-internal use), an exported object is given a unique number. This number assumes the place of the address in the symbol file. Typically, these numbers are allocated strictly sequentially.

As a consequence, when compiling a client, only module specific numbers are available, but no addresses. These numbers must, as mentioned before, be converted into absolute addresses upon loading. For this task, knowledge about the positions of such incomplete address fields must be available. Instead of supplying the object file with a list of all locations of such addresses, the elements of this fixup list are embedded in the instructions at the very places of the yet unknown addresses. This mirrors the technique used for the completion of addresses of forward jumps (see Chapter 11). If all such addresses to be completed are collected in a single fixup list, then this corresponds to the Figure 15.8 (a). Every element must be identified with a

pair consisting of a module number (mno) and an entry number (eno). It is simpler to provide a separate list for every module. In the object file, not just a single fixup root, but one for each list is required. This corresponds to the Figure 15.8 (b). Part (c) shows the extreme solution where a separate fixup list is specified for every imported object. Which of the three presented solutions is adopted, depends on how much information can be put into the place of an absolute address, by which it is ultimately replaced.

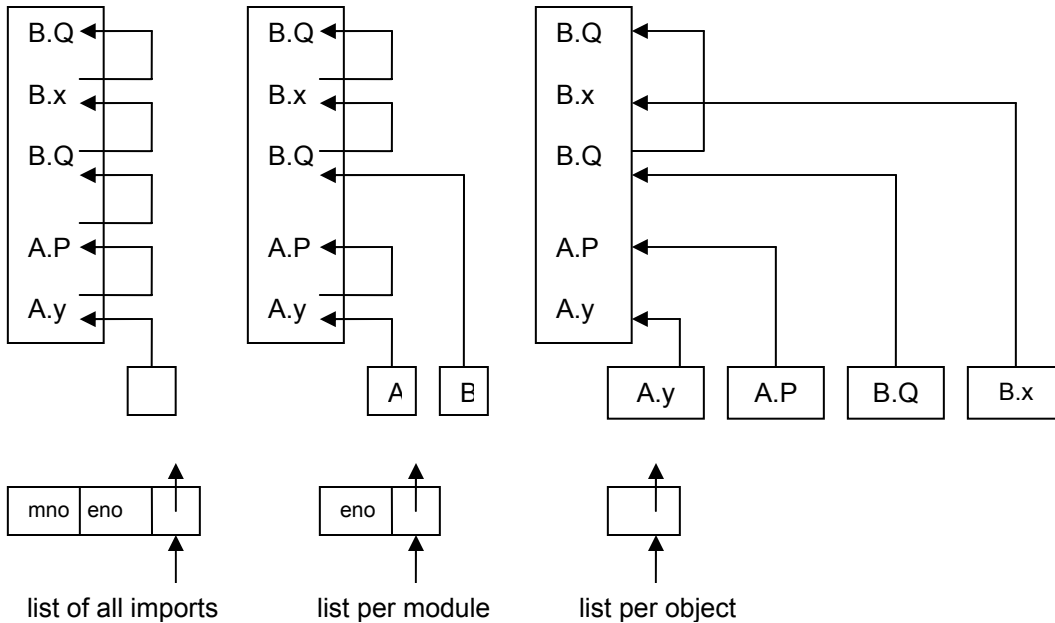


Figure 15.8. Three forms of fixup lists in object files.

15.5. Checking configuration consistency

It may seem belated if we now pose the question: Why are symbol files introduced at all? Let us assume that a module M is to be compiled which imports M0 and M1. A rather straightforward solution would be to recompile M0 and M1 immediately preceding the compilation of M, and to unite the three symbol tables obtained. The compilations of M0 and M1 might easily be triggered by the compilation of M reading the import list.

Although the repeated compilation of the same source text is a waste of time, this technique is used by various commercial compilers for (extended) Pascal and Modula. The serious shortcoming inherent in this method, however, is not so much the additional effort needed, but the lack of a guarantee for the consistency of the modules being bound. Let us assume that M is to be recompiled after some changes had been made in the source text. Then it is quite likely that after the original formulation of M and after its compilation, changes have also been made to M0 and M1. These changes may invalidate M. Perhaps even the source versions of M0 and M1 currently available to the programmer of client M no longer comply with the actual object files of M0 and M1. This fact, however, cannot be determined by a compilation of M, but it almost certainly leads to disaster when the inconsistent parts are bound and executed.

Symbol files, however, do not permit changes like source files; they are encoded and not visible through a text editor. They can only be replaced as a whole. In order to establish consistency, every symbol file is provided with a unique key. Symbol files thus make it possible to make modules available without giving away the source text. A client may rely on the specified interface definition and, thanks to the key, the consistency of the definition with the present implementations is also guaranteed. Unless this guarantee is provided, the entire notion of modules and separate compilation is perhaps enticing, but hardly a useful tool.

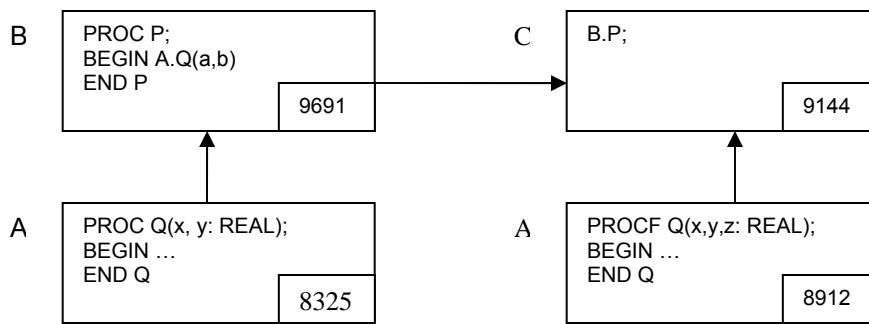


Figure 15.9. Inconsistency of module versions.

As an example, Figure 15.9 shows on its left side the situation upon compilation of module B, and on the right side that upon compilation of module C. Between the two compilations, A was changed and recompiled. The symbol files of B and C therefore contain module anchors of A with differing keys, namely 8325 in B and 8912 in C. The compiler checks the keys, notices the difference, and issues an error message. If, however, module A is changed after the recompilation of C (with changed interface), then the inconsistency can and must be detected upon loading and binding the module configuration. For this purpose, the same keys are also included in the respective object files. Therefore it is possible to detect the inconsistency of the import of A in B and C before execution is attempted. This is absolutely essential.

The key and the name are taken as the characteristic pair of every module, and this pair is contained in the heading of every symbol and object file. As already mentioned, the names of modules in the import list are also supplemented by their key.

Unique module keys can be generated by various algorithms. The simplest is perhaps the use of current time and date which, suitably encoded, yield the desired key. A drawback is that this method is not entirely reliable. Even if the resolution of the clock is one second, simultaneous compilations on different computers may generate the same key. Somewhat more significant is the argument that two compilations of the same source text should always generate the same key; but they do not. Hence, if a change is made in a module which is later detected to be in error, recompilation of the original version nevertheless results in a new key which lets old clients appear as invalidated.

A better method to generate a key is to use the symbol file itself as argument, like in the computation of a checksum. But this method is also not entirely safe, because different symbol files may result in the same key. But it features the advantage that every recompilation of the same text generates the same key. Keys computed in this way are called fingerprints.

15.6. Exercises

15.1. Incorporate separate compilation into your Oberon-0 compiler. The language is extended to include an import list and a marker in the exported identifier's declaration. Use the technique of symbol files and introduce the rule that exported variables may not be assigned values from outside, that is, in importing modules that are considered to be read-only variables.

15.2. Implement a fingerprint facility for generating module keys.

16. Optimizations and the Frontend/Backend Structure

16.1. General considerations

If we analyse the code generated by the compiler developed in the preceding chapters, we can easily see that it is correct and fairly straightforward, but in many instances also improvable. The reason primarily lies in the directness of the chosen algorithm which translates language constructs independently of their context into fixed patterns of instruction sequences. It hardly perceives special cases and does not take advantage of them. The directness of this scheme leads to results that are only partially satisfactory as far as economy of storage and execution speed are concerned. This is not surprising, as source and target languages do not correspond in simple ways. In this connection we can observe the semantic gap between programming language on the one hand and instruction set and machine architecture on the other.

In order to generate code which utilizes the available instructions and machine resources more effectively, more sophisticated translation schemes must be employed. They are called optimizations, and compilers using them are said to be optimizing compilers. It must be pointed out that this term, although in widespread use, basically is a euphemism. Nobody would be willing to claim that the code generated by them could be optimal in all cases, that is, in no way improvable. The so-called optimizations are nothing more than improvements. However, we shall comply with the common vocabulary and will also use the term optimization.

It is fairly evident that the more sophisticated the algorithm, the better the code obtained. In general it can be claimed that the better the generated code and the faster its execution, the more complex, larger and slower will be the compiler. In some cases, compilers have been built which allow a choice of an optimization level: while a program is under development, a low, and after its completion a high, degree of optimization is selected for compilation. As an aside, note that optimization may be selected with different goals, such as towards faster execution or towards denser code. The two criteria usually require different code generation algorithms and are often contradictory, a clear indication that there is no such thing as a well-defined optimum.

It is hardly surprising that certain measures for code improvement may yield considerable gains with modest effort, whereas others may require large increases in compiler complexity and size while yielding only moderate code improvements, simply because they apply in rare cases only. Indeed, there are tremendous differences in the ratio of effort to gain. Before the compiler designer decides to incorporate sophisticated optimization facilities, or before deciding to purchase a highly optimizing, slow and expensive compiler, it is worth while clarifying this ratio, and whether the promised improvements are truly needed.

Furthermore, we must distinguish between optimizations whose effects could also be obtained by a more appropriate formulation of the source program, and those where this is impossible. The first kind of optimization mainly serves the untalented or sloppy programmer, but merely burdens all the other users through the increased size and decreased speed of the compiler. As an extreme example, consider the case of a compiler which eliminates a multiplication if one factor has the value 1. The situation is completely different for the computation of the address of an array element, where the index must be multiplied by the size of the elements. Here, the case of a size equal to 1 is frequent, and the multiplication cannot be eliminated by a clever trick in the source program.

A further criterion in the classification of optimization facilities is whether or not they depend on a given target architecture. There are measures which can be explained solely in terms of the source language, independent of any target. Examples of target-independent optimizations are suggested by the following well known identities:

```
x + 0 = x
x * 2 = x + x
b & TRUE = b
b & ~b = FALSE
```

```
IF TRUE THEN A ELSE B END = A
IF FALSE THEN A ELSE B END = B
```

On the other hand, there are optimizations that are justified only through the properties of a given architecture. For example, computers exist which combine a multiplication and an addition, or an addition, a comparison and a conditional branch in a single instruction. A compiler must then recognize the code pattern which allows the use of such a special instruction.

Lastly, we must also point out that the more optimizations with sizeable effects that can be incorporated in a compiler, the poorer its original version must have been. In this connection, the cumbersome structures of many commercial compilers, whose origin is difficult to fathom, lead to surprisingly poor initial performance, which makes optimizing features seem absolutely indispensable.

16.2. Simple optimizations

First, let us consider optimizations that are implementable with little effort, and which therefore are practically mandatory. This category includes the cases which can be recognized by inspection of the immediate context. A prime example is the evaluation of expressions with constants. This is called constant folding and is already contained in the compiler presented.

Another example is multiplication by a power of 2, which can be replaced by a simple, efficient shift instruction. Shift operations are usually much faster than multiplication. Also this case can be recognized without considering any context:

```
IF (y.mode = Const) & (y.a # 0) THEN
    n := y.a; k := 0;
    WHILE ~ODD(n) DO n := n DIV 2; k := k+1 END ;
    IF n = 1 THEN Put1(LSL, R0, R0, k) ELSE Put1(MUL, R0, R0, y.a) END
ELSE ...
END
```

Division (of integers) is treated in the same way. If the divisor is 2^k for some integer k , the dividend is merely shifted k bits to the right. For the modulo operator, the least significant k bits are simply masked out.

16.3. Avoiding repeated evaluation

Perhaps the best known case among the target independent optimizations is the elimination of common subexpressions. At first sight, this case may be classified among the elective optimizations, because the re-evaluation of the same subexpression can be achieved by a simple change of the source program. For example, the assignments

```
x := (a+b)/c; y := (a+b)/d
```

can easily be replaced by three simpler assignments when using an auxiliary variable u :

```
u := a+b; x := u/c; y := u/d
```

Certainly, this is an optimization with respect to the number of arithmetic operations, but not with respect to the number of assignments or the clarity of the source text. Therefore the question remains open as to whether this change constitutes an improvement at all.

More critical is the case where the improvement is impossible to achieve by a change of the source text, as is shown in the following example:

```
a[i, j] := a[i, j] + b[i, j]
```

Here, the same address computation is performed three times, and each time it involves at least one multiplication and one addition. The common subexpressions are implicit and not directly visible in the source. An optimization can be performed only by the compiler.

Elimination of common expressions is only worth while if they are evaluated repeatedly. This may even be the case if the expression occurs only once in the source:

```
WHILE i > 0 DO z := x+y; i := i-1 END
```

Since x and y remain unchanged during the repetition, the sum need be computed only once. The compiler must pull the assignment to z out of the loop. The technical term for this feat is loop invariant code motion.

In all the latter cases code can only be improved by selective analysis of context. But this is precisely what increases the effort during compilation very significantly. The compiler presented for Oberon-0 does not constitute a suitable basis for this kind of optimization.

Related to the pulling out of constant expressions from loops is the technique of simplifying expressions by using the values computed in the previous repetition, that is, by considering recurrence relations. If, for example, the address of an array element is given by $\text{adr}(a[i]) = k*i + a_0$, then $\text{adr}(a[i+1]) = \text{adr}(a[i]) + k$. This case is particularly frequent and therefore relevant. For instance, the addresses of the indexed variables in the statement

```
FOR i := 0 TO N-1 DO a[i] := b[i] * c[i] END
```

can be computed by a simple addition of a constant to their previous values. This optimization leads to significant reductions in computation time. A test with the following example of a matrix multiplication showed surprising results:

```
FOR i := 0 TO 99 DO
  FOR j := 0 TO 99 DO
    FOR k := 0 TO 99 DO a[i, j] := a[i, j] + b[i, k] * c[k, j] END
  END
END
```

The use of registers instead of memory locations to hold index values and sums, and the elimination of index bound tests resulted in a speed increased by a factor of 1.5. The replacement of indexed addressing by linear progression of addresses as described above yielded a factor of 2.75. And the additional use of a combined multiplication and addition instruction to compute the scalar products increased the factor to 3.90.

Unfortunately, not even consideration of simple context information suffices in this case. A sophisticated control and data flow analysis is required, as well as detection of the fact that in each repetition an index is incremented monotonically by 1.

16.4. Register allocation

The dominant theme in the subject of optimization is the use and allocation of processor registers. In the Oberon-0 compiler presented registers are used exclusively to hold anonymous intermediate results during the evaluation of expressions. For this purpose, usually a few registers suffice. Modern processors, however, feature a significant number of registers with access times considerably shorter than that of main memory. Using them for intermediate results only would imply a poor utilization of the most valuable resources. A primary goal of good code optimization is the most effective use of registers in order to reduce the number of accesses to the relatively slow main memory. A good strategy of register usage yields more advantages than any other branch of optimization.

A widespread technique is register allocation using graph colouring. For every value occurring in a computation, that is, for every expression the point of its generation and the point of its last use are determined. They delimit its range of relevance. Obviously, values of different expressions can be stored in the same register, if and only if their ranges do not overlap. The ranges are represented by the nodes of a graph, in which an edge between two nodes signifies that the two ranges overlap. The allocation of N available registers to the occurring values may then be understood as the colouring of the graph with N colours in such a way that

neighbouring nodes always have different colours. This implies that values with overlapping ranges are always allocated to different registers.

Furthermore, selected, scalar, local variables are no longer allocated in memory at all, but rather in dedicated registers. In order to approach an optimal register utilization, sophisticated algorithms are employed to determine which variables are accessed most frequently. Evidently, the necessary bookkeeping about variable accesses grows, and thereby compilation speed suffers. Also, care has to be taken that register values are saved in memory before procedure calls and are restored after the procedure return. The lurking danger is that the effort necessary for this task surpasses the savings obtained. In many compilers, local variables are allocated to registers only in procedures which do not contain any calls themselves (leaf procedures), and which therefore are also called most frequently, as they constitute the leaves in the tree representing the procedure call hierarchy.

A detailed treatment of all these optimization problems is beyond the scope of an introductory text about compiler construction. The above outline shall therefore suffice. In any case such techniques make it clear that for a nearly optimal code generation significantly more information about context must be considered than is the case in our relatively simple Oberon-0 compiler. Its structure is not well-suited to achieving a high degree of optimization. But it serves excellently as a fast compiler producing quite acceptable, although not optimal code, as is appropriate in the development phase of a system, particularly for educational purposes. Section 16.5 indicates another, somewhat more complex compiler structure which is better suited for the incorporation of optimization algorithms.

16.5. The frontend/backend compiler structure

The most significant characteristic of the compiler developed in Chapters 7 _ 12 is that the source text is read exactly once. Code is thereby generated on the fly. At each point, information about the operands is restricted to the items denoting the operand and to the symbol table representing declarations. The so-called frontend/backend compiler structure, which was briefly mentioned in Chapter 1, deviates decisively in this respect. The frontend part also reads the source text once only, but instead of generating code it builds a data structure representing the program in a form suitably organized for further processing. All information contained in statements is mapped into this data structure. It is called a syntax tree, because it also mirrors the syntactic structure of the text. Somewhat oversimplifying the situation, we may say that the frontend compiles declarations into the symbol table and statements into the syntax tree. These two data structures constitute the interface to the backend part whose task is code generation. The syntax tree allows fast access to practically all parts of a program, and it represents the program in a preprocessed form. The resulting compilation process is shown in Figure 16.1.

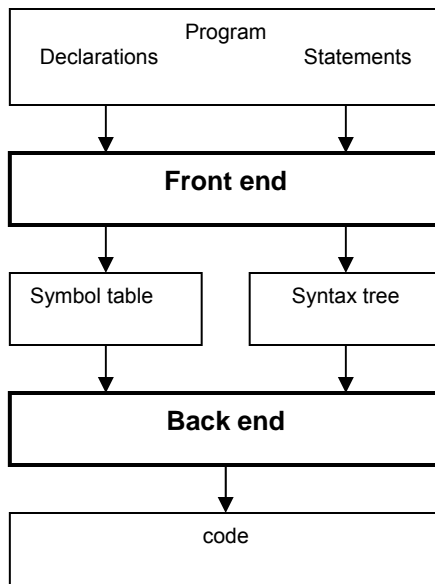


Figure 16.1. Compiler consisting of front end and back end

We pointed out one significant advantage of this structure in Chapter 1: the partitioning of a compiler in a target-independent front end and a target-dependent back end. In the following, we focus on the interface between the two parts, namely the structure of the syntax tree. Furthermore, we show how the tree is generated.

Exactly as in a source program where statements refer to declarations, so does the syntax tree refer to entries in the symbol table. This gives rise to the understandable desire to declare the elements of the symbol table (objects) in such a fashion that it is possible to refer to them from the symbol table itself as well as from the syntax tree. As basic type we introduce the type *Object* which may assume different forms as appropriate to represent constants, variables, types, and procedures. Only the attribute *type* is common to all. Here and subsequently we make use of Oberon's feature called *type extension* (Reiser and Wirth, 1992).

```

Object      = POINTER TO ObjDesc;
ObjDesc    = RECORD type: Type END ;
ConstDesc  = RECORD (ObjDesc) value: LONGINT END ;
VarDesc    = RECORD (ObjDesc) adr, level: LONGINT END ;
  
```

The symbol table consists of lists of elements, one for each scope (see Section 8.2). The elements consist of the name (identifier) and a reference to the identified object.

```

Ident      = POINTER TO IdentDesc;
IdentDesc  = RECORD name: ARRAY 32 OF CHAR;
             obj: Object; next: Ident
             END ;
Scope     = POINTER TO ScopeDesc;
ScopeDesc  = RECORD first: Ident; dsc: ScopeDesc END ;
  
```

The syntax tree is best conceived as a binary tree. We call its elements Nodes. If a syntactic construct has the form of a list, it is represented as a degenerate tree in which the last element has an empty branch.

```

Node =      POINTER TO NodeDesc;
NodeDesc =  RECORD (Object)
             op: INTEGER;
             left, right: Object
             END
  
```

Let us consider the following brief excerpt of a program text as an example:

```
VAR x, y, z: INTEGER;
BEGIN z := x + y - 5; ...
```

The front end parses the source text and builds the symbol table and the syntax tree as shown in Figure 16.2. Representations of data types are omitted.

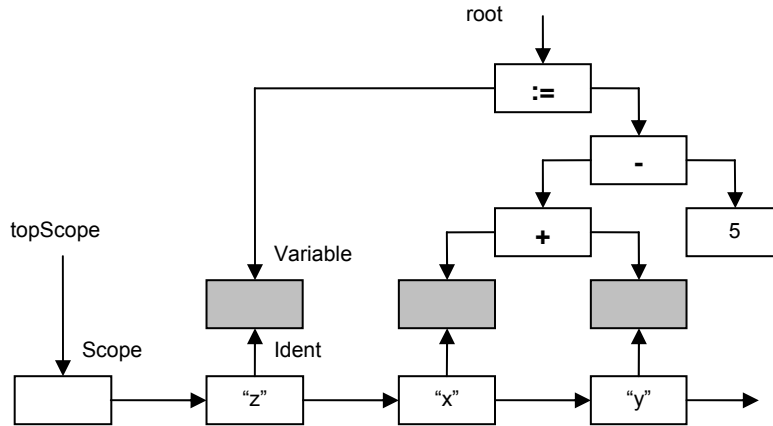


Figure 16.2. Symbol table (below) and syntax tree (above).

Representations of procedure calls, the IF and WHILE statements and the statement sequence are shown in Figures 16.3 - 16.5.

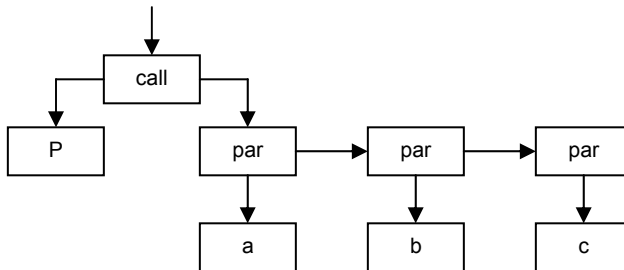


Figure 16.3. Procedure call.

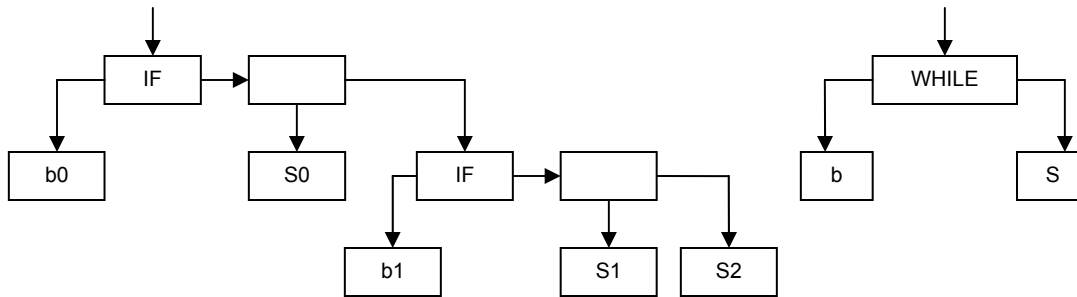


Figure 16.4. IF and WHILE statements.

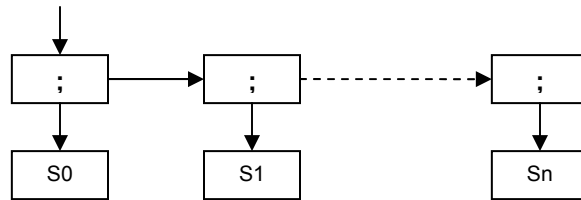


Figure 16.5. Statement sequence.

To conclude, the following examples demonstrate how the described data structures are generated. The reader should compare these compiler excerpts with the corresponding procedures of the Oberon-0 compiler. All subsequent algorithms make use of the auxiliary procedure *New*, which generates a new node.

```

PROCEDURE New(op: INTEGER; x, y: Object): Item;
  VAR z: Item;
BEGIN New(z); z.op := op; z.left := x; z.right := y; RETURN z
END New;

PROCEDURE factor(): Object;
  VAR x: Object; c: Constant;
BEGIN
  IF sym = ident THEN x := This(name); Get(sym); x := selector(x)
  ELSIF sym = number THEN NEW(c); c.value := number; Get(sym); x := c
  ELSIF sym = lparen THEN Get(sym); x := expression();
    IF sym = rparen THEN Get(sym) ELSE Mark(22) END
  ELSIF sym = not THEN Get(sym); x := New(not, NIL, factor())
  ELSE ...
  END ;
  RETURN x
END factor;

PROCEDURE term(): Object;
  VAR op: INTEGER; x: Object;
BEGIN x := factor();
  WHILE (sym >= times) & (sym <= and) DO
    op := sym; Get(sym); x := New(op, x, factor())
  END ;
  RETURN x
END term;

PROCEDURE statement(): Object;
  VAR x: Object;
BEGIN
  IF sym = ident THEN
    x := This(name); Get(sym); x := selector(x);
    IF sym = becomes THEN Get(sym); x := New(becomes, x, expression())
  ELSIF ...
  END
  ELSIF sym = while THEN
    Get(sym); x := expression();
    IF sym = do THEN Get(sym) ELSE Mark(25) END ;
    x := New(while, x, statseq());
    IF sym = end THEN Get(sym) ELSE Mark(20) END
  ELSIF ...
  END ;
  RETURN x
END statement

```

These excerpts clearly show that the structure of the front end is predetermined by the parser. The program has even become slightly simpler. But it must be kept in mind that type checking

has been omitted in the above procedures for the sake of brevity. However, as a target-independent task, type checking clearly belongs to the front end.

16.6. Exercises

16.1. Improve code generation of the Oberon-0 compiler such that values and addresses, once loaded into a register, may possibly be reused without reloading. For the example

$z := (x - y) * (x + y); y := x$

the presented compiler generates the instruction sequence

```
LDW      R0, SP, x
LDW      R1, SP, y
SUB      R0, R0, R1
LDW      R1, SP, x
LDW      R2, SP, y
ADD      R1, R1, R2
MUL      R0, R0, R1
STW      R0, SP, z
LDW      R0, SP, x
STW      R0, SP, y
```

The improved version is to generate

```
LDW      R0, SP, x
LDW      R1, SP, y
SUB      R2, R0, R1
ADD      R3, R0, R1
MUL      R4, R2, R3
STW      R4, SP, z
STW      R0, SP, y
```

Measure the gain on hand of a reasonably large number of test cases.

- 16.2. Which additional instructions of the RISC architecture of Chapter 9 would be desirable to facilitate the implementations of the preceding exercises, and to generate shorter and more efficient code?
- 16.3. Optimize the Oberon-0 compiler in such a way that scalar variables are allocated in registers instead of memory if possible. Measure the achieved gain and compare it with the one obtained in Exercise 16.1. How are variables treated as VAR parameters?
- 16.4. Construct a module OSG_x which replaces OSG and generates code for a CISC architecture *x*. The given interface of OSG should be retained as far as possible in order that modules OSS and OSP remain unchanged.

References

- A.V. Aho, J.D. Ullman. *Principles of Compiler Design*. Reading MA: Addison-Wesley, 1985.
- F. L. DeRemer. Simple LR(k) grammars. *Comm. ACM*, 14, 7 (July 1971), 453-460.
- M. Franz. The case for universal symbol files. *Structured Programming 14* (1993), 136-147.
- S. L. Graham, S. P. Rhodes. Practical syntax error recovery. *Comm. ACM*, 18, 11, (Nov. 1975), 639-650.
- J. L. Hennessy, D. A. Patterson. *Computer Architecture. A Quantitative Approach*. Morgan Kaufmann, 1990.
- C.A.R. Hoare. Notes on data structuring.
In *Structured Programming*. O.-J. Dahl, E.W. Dijkstra, C.A.R. Hoare, Acad. Press, 1972.
- U. Kastens. *Uebersetzerbau*. Oldenbourg, 1990
- D. E. Knuth. On the translation of languages from left to right.
Information and Control, 8, 6 (Dec. 1965), 607-639.
- D.E. Knuth. Top-down syntax analysis. *Acta Informatica 1* (1971), 79-110.
- W. R. LaLonde, et al. An LALR(k) parser generator.
Proc. IFIP Congress 71, North-Holland, 153-157.
- J.G.Mitchell, W. Maybury, R. Sweet. Mesa Language Manual.
Xerox Palo Alto Research Center, Technical Report CSL-78-3.
- P. Naur (Ed). Report on the algorithmic language Algol 60.
Comm. ACM, 3 (1960), 299-314, and *Comm. ACM*, 6, 1 (1963), 1-17.
- P. Rechenberg, H. Mössenböck. *Ein Compiler-Generator für Mikrocomputer*. C. Hanser, 1985.
- M. Reiser, N. Wirth. *Programming in Oberon*. Wokingham: Addison-Wesley, 1992.
- N. Wirth. The programming language Pascal. *Acta Informatica 1* (1971)
- N. Wirth. Modula - A programming language for modular multiprogramming.
Software - Practice and Experience, 7, (1977), 3-35.
- N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Comm. ACM*, 20, (1977), 11, 822-823.
- N. Wirth. *Programming in Modula-2*. Heidelberg: Springer-Verlag, 1982.
- N. Wirth and J. Gutknecht. *Project Oberon*. Wokingham: Addison-Wesley, 1992.

Appendix A

Syntax of Oberon-0

ident = letter {letter | digit}.

integer = digit {digit}.

selector = {"." ident | "[" expression "]"}.

factor = ident selector | integer | "(" expression ")" | "~" factor.

term = factor {"*" | "DIV" | "MOD" | "&"} factor.

SimpleExpression = ["+" | "-"] term {"+" | "-"} | "OR" term}.

expression = SimpleExpression
["=" | "#" | "<" | "<=" | ">" | ">="] SimpleExpression].

assignment = ident selector ":@" expression.

ActualParameters = "(" [expression {"," expression}] ")" .

ProcedureCall = ident [ActualParameters].

IfStatement = "IF" expression "THEN" StatementSequence
{"ELSIF" expression "THEN" StatementSequence}
["ELSE" StatementSequence] "END".

WhileStatement = "WHILE" expression "DO" StatementSequence "END".

RepeatStatement = "REPEAT" StatementSequence "UNTIL" expression.

statement = [assignment | ProcedureCall | IfStatement | WhileStatement | RepeatStatement].

StatementSequence = statement {";" statement}.

IdentList = ident {"," ident}.

ArrayType = "ARRAY" expression "OF" type.

FieldList = [IdentList ":" type].

RecordType = "RECORD" FieldList {";" FieldList} "END".

type = ident | ArrayType | RecordType.

FPSection = ["VAR"] IdentList ":" type.

FormalParameters = "(" [FPSection {";" FPSection}] ")" .

ProcedureHeading = "PROCEDURE" ident [FormalParameters].

ProcedureBody = declarations ["BEGIN" StatementSequence] "END".

ProcedureDeclaration = ProcedureHeading ";" ProcedureBody ident.

declarations = ["CONST" {ident "=" expression ";"}]
["TYPE" {ident "=" type ";"}]
["VAR" {IdentList ":" type ";"}]
{ProcedureDeclaration ";"}

module = "MODULE" ident ";" declarations
["BEGIN" StatementSequence] "END" ident "." .

Appendix B

The ASCII Character Set

.	0	1	2	3	4	5	6	7
0	nul	dle		0	@	P	`	p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	\$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
8	bs	can	(8	H	X	h	x
9	ht	em)	9	I	Y	i	y
A	lf	sub	*	:	J	Z	j	z
B	vt	esc	+	;	K	[k	{
C	ff	fs	,	<	L	\	l	
D	cr	gs	-	=	M]	m	}
E	so	rs	.	>	N	^	n	~
F	si	us	/	?	O	_	o	del