An Oberon Compiler for the ARM Processor

Niklaus Wirth

18.12.2007, rev. 16.4.2008

0. Table of Contents

1. History and Introduction

In late 1997 we decided to program the control system for a model helicopter in the language Oberon. At the same time, the decision was made to use DEC's StrongARM processor DS1035 as the core of the system. An immediate consequence was the necessity of an Oberon compiler for that processor and, since none was available, to build one. An important objective was to demonstrate the language's suitability to express programs for real-time applications, requiring efficient and predictable performance, and encouraging structured and modular design.

Although compiler technology is a reasonably mature subject, most engineers hesitate to build their own compilers, particularly if manpower is scarce. The principal reason is the belief that compilers are inherently complex and large programs. At least since the advent of Reduced Instruction Set Computers (RISC), there is good reason for this belief. The credo of this technology is that computer architectures must concentrate on the basics, and compilers would do the rest, irrespective of complications. We believe that also languages should concentrate on the essentials and avoid complexity through regularity.

Instead of choosing the usual path of using tools for syntactic analysis and for code generation as black boxes, I decided to build the compiler alone and from scratch. This was not unreasonable in view of my previous experience in compiler construction. Certainly, the goal of achieving highly optimized code had to be abandoned. Straight-forward code generation would have to do. A first version of the compiler was to be available within a few months. Hence, I decided to build the compiler along the lines of a simple single-pass compiler with on-the-fly code generation as elaborated in [1]. Having a scanner and a parser already available saved a few days of labor. But the crucial part, without doubt, is code generation. We also decided to implement a subset of Oberon, and to drop features that seemed superfluous for the task at hand.

Therefore, we will report on the conventional part, the scanner and the parser, only scantily. Instead, we discuss in more detail our approach to achieving the efficiency and economy that an engineer building a real-time control system demands. He must be able to rely on predictable performance, a requirement more fundamental than mere efficiency. Modern processors do not perform very well in this respect. Instruction pipelines and caches speed up computations considerably, but still yield worst cases that let this gain be of doubtful value. Sophisticated compilers, rearranging and contorting code in obscure ways add to this misery. In contrast, a compiler that produces code according to a few, obvious rules is a preferable solution. We feared that a suboptimal performance of the generated code might be a severe handicap. We even introduced a feature that was specifically tailored to the architecture to produce efficient, sequential array access. In hindsight, the fear of insufficient performance was unjustified. Even the special speed-feature was dropped. Instead, the great speed of the compilation process is an invaluable gain in a development environment.

More recently, I added a third goal, namely to separate the language-dependent and target-independent parts, such as scanning and parsing, more systematically from the language-independent and target-dependent parts, such as code generation. If at all possible, they should be separated into different modules. It was in particular this third aim that increased the amount of work. In fact, it caused me to write a practically new compiler.

The main features that had been omitted from the subset and now had to be added, were type extensions (inheritance) with type test and type guards, complete index range checking for open arrays, the type SET and its operators, the case statement, and various other features, such as copying and comparing arrays, strings, and import/export of variables.

A revision of Oberon as defined in 1988 was issued in 2007, with several restrictions and eliminations in the sense of a cleanup. We here summarize the main changes of the revision called Oberon-07:

1. No access to intermediate-level objects.
2. The basic types SHORTINT and LONGINT are eliminated, and with it type inclusion.
3. The loop and exit statement are eliminated.
4. The with statement is eliminated.
5. Variables are exported in read-only mode.
6. Case labels must be non-negative integers, like array indices.

We also summarize the main features that were *added* and therefore must be considered as language *extensions* particular for this implementation.

1. Leaf procedures and register variables
2. Interrupt handlers

2. The Structure of the Compiler

The compiler is structured conventionally and conveniently into modules according to the major tasks of scanning the source text, parsing and type checking, and code generation. In addition, a module is provided containing the definition of data types used throughout the entire compiler. These are in particular the types *Object*, denoting elements of the “symbol table”, and *Type*, describing the types of data and functions. This module also contains routines for constructing and for searching the data structure. More significantly, it also contains the machinery for generating and loading symbol files for separate, type checked compilation of modules.

Module	Data owned	source lines	chars	code (bytes)
Parser (OSAP)	(command module)	1000	40K	21.5K
Generator (OSAG)	type *Item*, target code, registers	1400	50K	26.8K
Base (OSAB)	types *Object* and *Type*, symbol table	440	16K	7.7K
Scanner (OSAS)	Source text, table of keywords	280	10K	5.8K
Total		3120	116K	61.8K

The parser imports from all other modules, in particular the symbols from the scanner, the basic types from the base, and the code generator procedures from the Generator. The generator in turn imports again the types from the base, and from the scanner the procedure *Mark* to report errors. For further explanations, we refer to [Wirth95, Compiler Construction, Wirth95, Project Oberon].

3. The Scanner (OSAS)

The task of the scanner is the recognition of language symbols (tokens) in the source text. The tokens are identifiers, integers, real numbers, strings, and other symbols. The latter are either special characters, character pairs, or keywords:

+	:=	ARRAY	IMPORT	THEN
-	^	BEGIN	IN	TO
*	=	BY	IS	TRUE
/	#	CASE	MOD	TYPE
~	<	CONST	MODULE	UNTIL
&	>	DIV	NIL	VAR
.	<=	DO	OF	WHILE
,	>=	ELSE	OR	
;	..	ELSIF	POINTER	
|	:	END	PROCEDURE	
(	)	FALSE	RECORD	EXIT*
[	]	FOR	REPEAT	LOOP*
{	}	IF	RETURN	WITH*

(The symbols marked with an asterisk are not used in Oberon-SA, but they are nevertheless recognized by the scanner as keywords).

The keywords are contained in a hash table. The hash function is chosen such that most keywords are recognized in the first try, and only a few require two tries. The source text is passed as a parameter to the scanner by calling *Init*. The source file is then attached to the rider *R* which keeps track of the current reading position. There is no backtracking. Procedure *Get* delivers the next symbol in the source text. Procedure *Mark* is used to issue error messages which refer to the current scanning position. Get and *Mark* are the important operators of this scanner, which shows a remarkably thin interface.

The global variables *ival, rval, slen, id,* and *str* are what might be called secondary results of procedure *Get*, when the last symbol delivered was an integer, a real number, a character or a string. *slen* indictes the number of characters in the string (including the terminating null character).

Further global variables are the reader *R* and the writer *W*, representing the source code and the diagnostic output. The latter is placed in the scanner module in order to correlate the diagnostic messages with the position reached in the source text.

Apart from all the language symbols, the scanner exports the following entities:

```
CONST IdLen = 32;

TYPE Ident = ARRAY IdLen OF CHAR;

VAR ival, slen: LONGINT;  (*attributes of symbol just read*)
      rval: REAL;
      error: BOOLEAN;
      id: Ident;  (*for identifiers*)
      str: ARRAY 60 OF CHAR;  (*for strings*)
      errcnt: INTEGER;

PROCEDURE Copy(VAR ident: Ident);
PROCEDURE Mark(msg: ARRAY OF CHAR);
PROCEDURE Get(VAR sym: INTEGER);
PROCEDURE Init(T: Texts.Text; pos: LONGINT);
```

The scanner skips comments, sequences of characters enclosed by comment brackets. Comments may be nested. Nesting is handled by a recursive comment recognizer. One anomaly in the language is the symbol ".." used in set denotations such as {5 .. 9}. It would require a look-ahead of more than one character to avoid conflicts with the decimal point in numbers. In order to retain the one-character look-ahead scheme, it is required that a space must lie between an integer and a ".." symbol.

The scanner is one of the few routines where a case statement is highly valuable.

4. The Parser (OSAP)

The parser is the "main module" of this compiler. It exports only the (parameterless) command *Compile* that can be activated by a click in any text. According to the Oberon system's conventions, it comes in three variants for indicating the text to be compiled:

OSAP.Compile @	most recently selected text
OSAP.Compile *	text in marked viewer
OSAP.Compile P0.Mod P1.Mod ……. Pn.Mod~	

The syntax of Oberon has been carefully designed such that texts can be parsed by the simple method of top-down parsing with a single symbol look-ahead. (Note that the total look-ahead is one symbol (by the parser) plus one character (by the scanner). For parsing we use the technique of recursive descent. Essentially every non-terminal class is represented by a parsing procedure. They are, in descending order:

```
Module
ProcedureDecl
Declarations
Type
Signature
FPSection  (Formal Parameter Section)
FormalType
RecordType
ArrayType
IdentList
StatSequence
expression
SimpleExpression
term
factor
set
element
ParameterList
Parameter
selector
TypeTest
```

Note that identifiers, numbers, strings are considered as "terminal" symbols and are recognized and delivered by the scanner.

While parsing declarations, a data structure conventionally called “the symbol table” is constructed. It is the dictionary of all declared identifiers together with the attributes of the objects introduced. The presence of the table is necessary for type consistency checking, beside parsing the second important task of this module.

This structure, however, is not a simple table, but a complex graph with its elements linked by pointers. It contains two types of elements: *Objects* represent *named*, declared entities, such as constants, variables, procedures etc. *Types* represent the types of objects. Note that many objects may be of the same type, and hence a representation using pointers is highly appropriate in this case. For details we refer to [2].

The entities declared in a scope are represented in a linked list (field *next*) rather than a tree. Its first element is a header. The headers of open scopes are also linked (*anc*) and form a pulsating stack. See procedures *OpenScope* and *CloseScope* in module OSAB..

Objects are of various classes, distinguished by the record field *class*. Likewise *Types* are distinguished by their form, represented by the field *form*. The possible values of these attributes are listed below. *Types* contain a field *base*. Its meaning is indicated below:

Classes

1	Const	constants
2	Var	variables (or value parameters)
3	Par	(reference) parameters
4	Field	record fields
5	Type	types
6	SProc	predefined, in-line procedures and functions
7	Mod	modules
8	Reg	register variables
9	RegI	(reference) parameters in registers

Forms

1	Byte	byte	(available as formal type SYSTEM.Byte)
2	Bool	Boolean	
3	Char	character	
4	Int	integer	
5	Real	real number	
6	Set	set	
7	Pointer	pointer	base = type of referenced record
8	NilType		
9	NoType		e.g. for “result” of proper procedures
10	Proc	procedure	base = result type
11	String	string	
12	Array	array	base = element type
13	Record	record	base = type of which this type is an extension

The declarations of *Object* and *Type* are defined in the service module OSAB (B for base). The same holds for the constants identifying classes and forms as shown above.

```
Name = ARRAY NameLen OF CHAR;
Object = POINTER TO ObjDesc;
Type = POINTER TO TypeDesc;

ObjDesc = RECORD
    class, lev, expo: INTEGER;
    rdo: BOOLEAN;   (*read only*)
    next, anc: Object;
    type: Type;
    name: Name;
    val: LONGINT
END ;

TypeDesc = RECORD
    form, ref: INTEGER;  (*ref is only used in exporting*)
    nofpar: INTEGER;  (*for procedures; extension level for records*)
    len: INTEGER;  (*for arrays, len < 0 => open*)
    dsc, typobj: Object;
    base: Type;  (*for arrays, records, pointers*)
```

```
      size: LONGINT  (*in bytes; always multiple of 4, except for Bool and Char*)
   END
```

While processing declarations of variables, addresses are associated with them, that is, storage is allocated. One might consider this as belonging into the module which contains all target-dependent matters. However, since allocation occurs in a strictly sequential manner, this task is machine-independent, at least if we consider byte-addressing a common standard. The running address is the global variable *dc* (data counter).

The field (attribute) *lev* denotes the level of nesting of the procedure (scope), in which the object is declared. Level 0 contains the global, statically allocated objects. *Expo* specifies whether or not the object is exported, *name* is the identifier denoting the object, *type* is the pointer referencing the object's type, and *val* denotes the object's value, if it is a constant, or its address, if it is a variable or procedure. Procedures are considered as constants of a procedure type.

In the case of *Types*, the attribute *nofpar* specifies the number of parameters of a procedure type, or the extension level in the case of records. *Len* indicates the length (number of elements) of an array. The value -1 is used to indicate that the array is open, i.e. that the length is unknown. In the case of a record type, *len* denotes the address of the corresponding type descriptor (see type extensions). The field *dsc* (descendant) points to the list of fields in the case of a record type, or to the list of parameters in the case of a procedure.

The attribute *size* indicates the number of bytes that an object of this type occupies. It is always a multiple of 4, except for the basic types *Boolean* and *Char*, where the size is 1. The field *ref* is used only while constructing the symbol file for exporting (see Ch. 24). The data structure corresponding to the following set of declarations is shown in Fig. 1.

```
TYPE R = RECORD f, g: INTEGER END ;
VAR x: INTEGER;
    a: ARRAY 10 OF INTEGER;
    r, s: R;
```

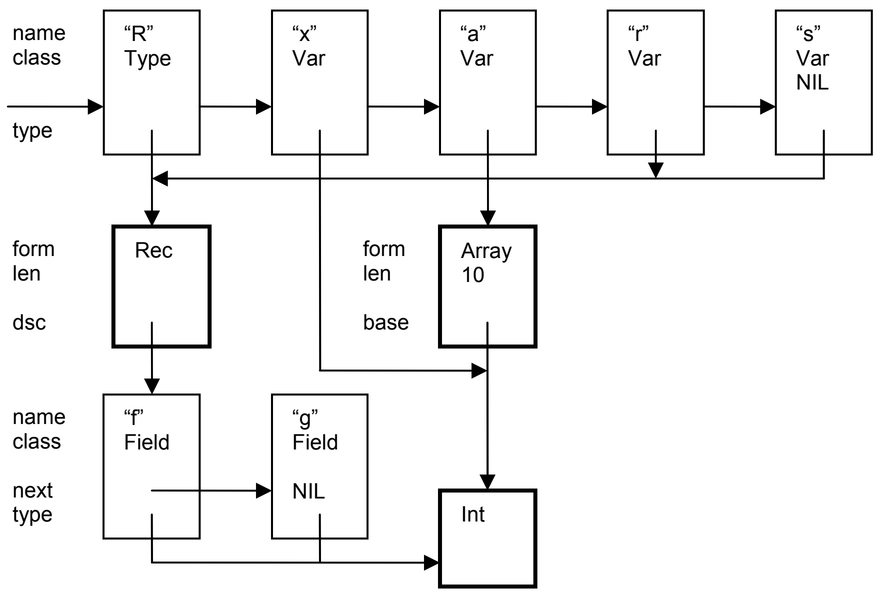


Fig. 1 Example of a "symbol table"

The parser module also performs the important task of *type consistency checking*, which, like parsing itself, is target code independent. It is the symbol table, retaining the properties of the declared objects, that makes type checking possible.

All parsing procedures carry one or more parameters of type *Item*. These procedures not only parse the text, but also deliver an encoded description of the parsed construct, such as an expression, in the form of the resulting *Item* record. The type *Item* is closely related to the type *Object*. Like an *Object*, an *Item* specifies its type, and with it the parser performs type checking. In contrast to an object, an item has no name, but other attributes used for code generation on-the- fly. The type is defined in module OSAG, as are all procedures operating on items. Details are explained in subsequent chapters on expression evaluation.

A very essential difference between *Objects* and *Items* is that the latter are never referenced via pointers, but are strictly bound to procedures and therefore reside on the processing stack. This “detail” significantly enhances the efficiency of the compiling process, because no storage management is required for *Items*. Whereas the “symbol table” with its Objects is constructed during the processing of declarations, and thereafter remains unchanged, Items rapidly come and go during the processing of statements and expressions. This is where efficiency counts.

The global variables *check, leaf,* and *int* are state variables indicating whether array bound checks should be generated or omitted, and whether a normal, a leaf, or an interrupt procedure is being compiled.

5. The ARM-Architecture

The ARM-architecture represents closely the classic Reduced Instruction Set Computer with an Arithmetic Logic Unit and an array of 16 general-purpose registers at its core. The ALU performs logical operations (not, and, or, xor) and arithmetic operations (add, subtract, multiply, but not divide, on integers). The second operand of the ALU can be selected from a register or from the instruction (immediate operand). A unique and most useful feature is the barrel shifter. It allows the second operand to be arbitrarily shifted before entering the ALU. This is highly convenient, because address computations often require shifts by powers of 2. Register 15 is the program counter (PC).

The instructions are classified according to their format and function as follows:

0. Register operations:	op, d, s0, s1, sc, md	R[d] := R[s0] op shift(R[s1], sc, md)
1. Register immediate:	op, d, s, imm	R[d] := R[s] op imm
2. Load/store:	op, r, b, a	R[r] := M[R[b]+a]; M[R[b]+a] := R[r]
3. Load/store:	op, r0, r1, r2, sc, md	R[r0] := M[R[r1]+shift([R[r2], sc, md)]
		M[R[r1]+shift([R[r2], sc, md)] := R[r0]
4. Load/store multiple:	op, r, set	sets of registers from/to memory
5. Branch	condition, offset	PC := PC + offset, if condition
6. Miscellaneous, including traps		

The basic ALU operations include copying, arithmetic and logical operations. The shift is determined by a shift count *sc* between 0 and 31 and a shift mode *md* (logical left, logical right, right with sign extension, and rotate right. All instructions may optionally set the *condition code* in the program status register (PSR). Of interest are the *Z*-bit, indicating whether a result is 0, and the *N*-bit, reflecting its sign bit. Unfortunately, the condition code is not set by instructions loading from memory. Fig. 2 shows a block diagram of ALU and Control Unit.

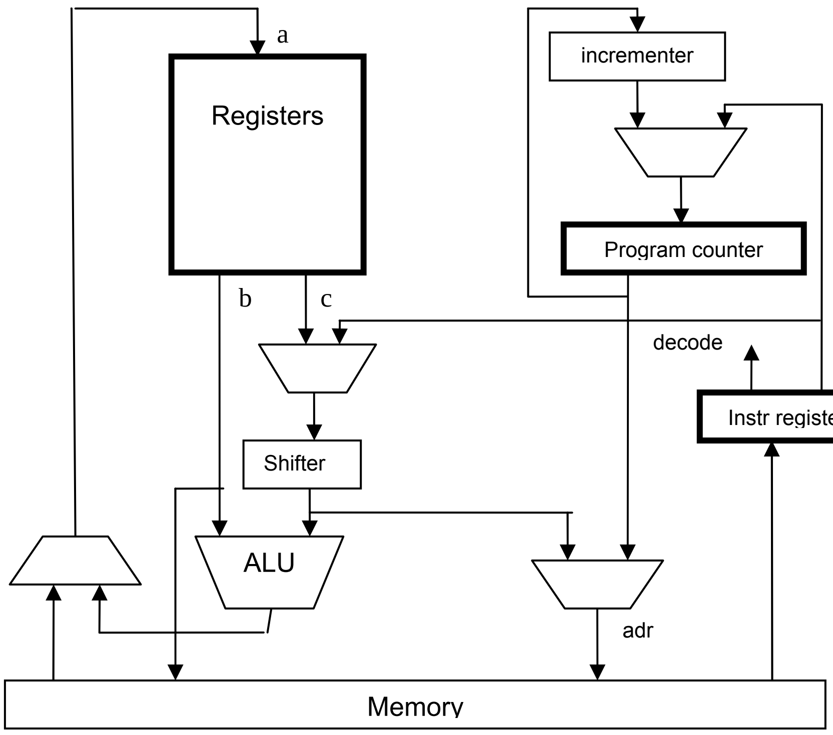


Fig. 2. Architecture of the ARM-core

True to the RISC-strategy, there exists only one addressing mode. The address is an offset added to a (base) register. The offset can be either a constant (immediate field) or a second register. In combination with the facility to increment or decrement this register before or after the memory access, this option is most useful for procedure calls (pushing parameters on the stack) and for copying and comparing arrays and strings (see the respective sections).

One of the unpleasant properties of this architecture is the shortness of the field for immediate values. It is only 8 bits long, and thus can specify values between 0 and 255 only. In fact, the field is 12 bits long, but 4 bits are used for a rotation count. The actual value is thus determined by the two fields *n* and *x* as "x rotated by 2n bits". This nasty feature appears as a mistake when one realises that every instruction has a condition field which is rarely used except for branch instructions. The same unpleasantness appears in load/store instructions, where the offset field is only 12 bits long. Moreover, two bits are "lost", because data come in words, hence having addresses which are multiples of 4, i.e. with two low-order zero bits (except for byte-data).

Branch instructions, on the other hand, feature a large, PC-relative offset field of 24 bits. Register R_{15} is the program counter. For procedure calls, a *branch and link* instruction is used that deposits the current PC in register R_{14}. Upon return, that value is moved back into R_{15} causing a return jump. In our implementation register R_{13} is used as the stack pointer SP, and R_{12} as the frame pointer FP. As a result, only registers R_0 – R_{11} are available for computations on data, which proves to be quite sufficient.

6. Storage Layout, Variable Allocation, and Procedure Calls

Every module occupies a storage block for its global (static) variables and, following it, a block for the program code and for constants. Together these blocks are said to constitute a *module frame*. A separate stack constitutes the workspace, a pulsating stack of *procedure frames* containing the local variables and parameters (see Ch. 16).

Variables and parameters are allocated sequentially in their respective block with negative offsets. The running counter is *dc*, a global variable of module OSAP, which apart from parsing also handles data space allocation. Global variables are addressed relative to PC, local variables and parameters relative to FP. (In the case of globals, this often leads to large offsets, which are complicated to represent in the ARM-architecture). Note that based addresses need not be changed during program loading. In fact, the loader affects only cross-module references.

Code is allocated in the global array *code* in module OSAG. The running index is *pc*, also global in OSAG. The format of instructions is represented by the various *Put* procedures, one or even several for each instruction class. Their parameters represent the instruction fields.

```
PROCEDURE Put0(op, dst, src0, src1: LONGINT);   (*register operation*)
PROCEDURE Put0a(op, dst, src0, src1, shmd, shcnt: LONGINT);   (*register operation with shift*)
PROCEDURE Put0b(op, dst, src0, src1, shmd, shreg: LONGINT);   (*register operation with shift cnt from reg*)
PROCEDURE Put0c(op, dst, src0, src1, src2: LONGINT);  (*multiply*)
PROCEDURE Put1(op, dst, src, imm: LONGINT);  (*register operation with immediate*)
PROCEDURE Put1a(op, dst, src, imm, rot: LONGINT);  (*register operation with rotated immediate*)
PROCEDURE Put2(op, reg, base, offset: LONGINT);  (*Load/store with offset literal*)
PROCEDURE Put3(op, reg, base, offreg, shift: LONGINT);  (*Load/store with offset from register*)
PROCEDURE Put4(op, base: LONGINT; regs: SET);  (*Load/Store multiple*)
PROCEDURE Put5(cond, offset: LONGINT);  (*Branch conditional*)
PROCEDURE Put5a(offset: LONGINT);  (*Branch and Link*)
PROCEDURE Put6(cond, num: LONGINT);  (*SWI conditional*)
```

Oberon-SA features the special facility of *register-variables*. Procedures can be explicitly designated as *leaf-procedures*, if they do not invoke other procedures, i.e. are leafs in the tree of calls. In this case, variables of types INTEGER and SET are automatically allocated in registers rather than in memory. This speeds up their access. The register bank is thus divided into three parts, with *regs* and *RH* being global variables in OSAG, The set variable *regs* indicates which registers are allocated for intermediate results in expressions.

R[0] … R[RH-1]	registers for temporary results
R[RH] … R[11]	parameters and register variables
R[12] … R[15]	dedicated registers (FP, SP, LNK, PC)

Every call of a procedure (except in-line procedures) generates a new frame in the stack where local variables are allocated. Its size is determined by the local declarations. This implies that variables are addressed by offsets relative to a frame address that is held in a register for reasons of efficiency. This register is called the *frame pointer* (FP). Global variables, however, have fixed, static addresses. In order to avoid the need for another base register, addresses of global variables are offsets relative to the program counter (PC), i.e. relative to the instruction addressing the global variable. Further details about procedure calls and storage layout are explained in section 16.

7. Constant- and Address Generation

Constants are placed in the immediate field of instructions if possible. However, this field is unfortunately only 8 bits wide, and hence, although the vast majority of constants lies between 0 and 255, requires that constants may be placed in memory and be accessed by load instructions. Again unfortunately, also load instruction feature a short offset field (12 bits). Thus “outplaced” constants are allocated following the end of each procedure. The immediate field is augmented by a rotation count field (4 bits). Hence, certain constants can be placed into the immediate field of instructions, even if their value is larger than 255.

Strings are insofar exceptional cases, as they are structured constants and may require several words of storage. Therefore, they are always “outplaced”. Strings are terminated by a null character and are always allocated in an integral number of words. (see procedures *enterIC, enterStr, ScaleConst*, and *FixupConstants* in OSAG).

The short offset fields in ARM instructions are a nasty problem for the compiler designer. The lesson for the programmer is that none of his procedures should become very long. The compiler designer, however, has to take care also of the unpleasant albeit rare cases. He resorts to the disliked solutions of either composing offsets through multiple instructions or through indirect addressing (procedures *Put1* and *Put2*).

The following example shows a source text and the generated code, as explained above. (Note: Branch instructions have PC-relative word offsets, and the PC is always 2 words ahead)

```
MODULE M;
  VAR a, b: INTEGER;                          a: -4, b: -8
  PROCEDURE P(x, y: INTEGER);                 x: -4, y: -8
    VAR z: INTEGER;                           z: -12
  BEGIN z := x + a
  END P;
BEGIN P(5, 256); P(-10, 53248)
END M.

  4   E51CB004     LDR    R11 FP -4           x  (local)
  5   E51FA020     LDR    R10 PC -32          a  (global)
  6   E09BB00A     ADD    R11 R11 R10         x+a
  7   E50CB00C     STR    R11 FP -12          z  (local)

 12   E3A0B005     MOV    R11  5
 13   E3A0AC01     MOV    R10  256
 14   EBFFFFF1     BSR    -15                 call at 1

 15   E3E0B009     MVN    R11  -10
 16   E3A0AA0D     MOV    R10  53248          [0000D000]
 17   EBFFFFEE     BSR    -18                 call at 1
```

8. Operands and Selectors

The parsing process has decomposed complex expressions into subexpressions consisting of an operator with its operands. We may therefore restrict our attention to single operators and their operands. The simplest case is when (both) operands are constants. Then the operation is performed by the compiler. This, however, is here done only for integer expressions. In all other cases we may consider the operation as being deferred to “execution-time”, as being replaced by the emission of instructions that later perform the operation. A reason for this is to avoid rarely beneficial complications of the compiler, another is to avoid overflow traps during compilation.

During the evaluation of expressions, intermediate results appear. For example, in (x+y) * (x-y) there exist two intermediate results, x+y, and x-y. They are also “objects”, but are anonymous. They pop up and disappear in a first-in-last-out order, and they are local to the procedure that parses them (*factor, term, simple expression, expression*). As explained above, they are similar to objects, but have no name. We call them *Items*. All parsing procedures in expressions carry items as parameters:

```
TYPE Item = RECORD
    mode: INTEGER;
    type: OSAB.Type;
    a, b, r: LONGINT;
    rdo: BOOLEAN  (*read only*)
END ;
```

The attribute *mode* corresponds to the attribute *class* of *Objects*. The meaning of fields *r, a, b* (hidden in OSAG) depends on the item’s mode: Together they hold the location (address) of the item. It is noteworthy that the meaning of the attribute *class* now transforms itself into that of an addressing *mode* with the correspondences indicated in the following table:

Mode	r	a	b	adr mode	
Const	-	value		immediate adr	value = a
Var	base	offset		direct adr	adr = R[r] + a
Par	base	offset0	offset1	indirect adr	adr = Mem[R[r]+a] + b
Reg	regno			register	value = R[r].

During the processing of an expression, all operands are represented by items. These undergo various state transitions with the ultimate destination of the mode *Reg,* because it is the characteristic of RISCs that operations are always performed on registers.

Objects are created from declarations. During the processing of statements, they are transformed into items by procedure *MakeItem*. These are objects representing constants, variables and parameters with the modes *Const, Var*, and *Par*. The mode transitions are accompanied by the emission of instructions, and they may be straightforward as in the case of simple variables, or complex, if selectors are present, such as indices, record field selectors, or dereferencing operations. In these cases, procedures *Index, Field*, and *Deref* (in OSAG) perform additional mode transformations and emissions of instructions. In this course, additional modes may arise that are not determined by language constructs, but rather by the target computer's addressing modes. In the case of the ARM architecture, there are only few of them, namely *RegI, RegX, CC, and Stk*:

Mode	r	a	b		
RegI	regno	offset		(register indirect)	adr = R[r] + a
RegX	reg0	reg1	scale	(register indirect)	adr = R[r] + R[a] * 2↑b
CC	cond	Fchain	Tchain	(condition code)	
Stk	SP			(top of stack)	adr = SP (push, pop)

The special mode *CC* signifies that the denoted value is in the special register *CC* that is part of the PSR. This will need special attention when explaining the processing of relations and Boolean operators. Note that the modes *Var* and *RegI* are actually identical. The difference is only that in the former case the register field *r* denotes a base register, *PC* or *FP*, whereas in the latter part *r* denotes any work register. Items with modes *Reg, CC,* and *Const* may also result from operators and relations.

The following procedures (in OSAG) take part in the transformation of items:

```
PROCEDURE MakeConstItem(VAR x: Item; typ: OSAB.Type; val: LONGINT);
PROCEDURE MakeRealItem(VAR x: Item; val: REAL);
PROCEDURE MakeStringItem(VAR x: Item; VAR s: ARRAY OF CHAR);
PROCEDURE MakeItem(VAR x: Item; y: OSAB.Object);

PROCEDURE Field(VAR x: Item; y: OSAB.Object);   (* x := x.y *)
PROCEDURE Index(VAR x, y: Item);   (* x := x[y] *)
PROCEDURE DeRef(VAR x: Item);   (* x := x^*)
```

The first group transforms (copies) *Objects* into *Items*, the latter transform items from one state to another while generating code. This will now be explained in further details with examples. Because an operand must be in a register in order that an arithmetic or logical operation can be applied, we first show how an operand is transferred into a register. This is the task of procedures *load(x)*, *loadAdr(x)*, and *loadCC(x).*

For each such procedure its task is described by the set of mode transitions and issued instructions. For simple variables the corresponding item is in mode *Var*. Before applying an operator, the item is loaded into a free register *r'* by procedure *load*:

mode	args		result	args	emitted instructions
Const	a	→	Reg	r'	MOV r', a
Var	r, a	→	Reg	r'	LDR r', [r, a]
Par	r, a, b	→	Reg	r'	LDR r', [r, a]; LDR r', [r', 0]

Consider the following example. Here, the reader needs to look at the program of module OSAG:

```
MODULE M;
   PROCEDURE P(x: INTEGER; VAR y: INTEGER);
      VAR z: INTEGER;
   BEGIN z := 10; z := (x + y) * (x - y)
   END P;
END M.
```

mode	args	result	args	emitted instructions	issued by

Const	10		Reg	11	MOVI	R11 10	load
Var	FP, -12	→			STR	R11 FP -12	Store
Var	FP, -4	→	Reg	11	LDR	R11 FP -4	load
Par	FP, -8	→			LDR	R10 FP -8	load
			Reg	10	LDR	R10 R10 0	load
			Reg	11	ADD	R11 R11 R10	AddOp
Var	FP, -4	→	Reg	10	LDR	R10 FP -4	load
Par	FP, -8	→			LDR	R9 FP -8	load
			Reg	9	LDR	R9 R9 0	load
			Reg	10	SUB	R10 R10 R9	AddOp
			Reg	11	MUL	R11 R11 R10	MulOp
Var	FP, -12				STR	R11 R12 -12	Store

Registers must be allocated in a strict stack discipline. This is necessary because parameter lists are built up in the register stack and released when the call instruction is issued. Allocation of a fresh register for an intermediate result would be straightforward, were it not for the facility of register variables. A typical case is the evaluation of, e.g. *x+y*. Register bookkeeping must release the registers holding operands *x* and *y*, and then allocate a register for the sum. Release must be supressed, however, if the operand is a register variable or a system register. The following auxiliary procedures serve as convenient abbreviations:

Release(u); GetReg(r)	→	GetReg1(r, u)
Release(v); Release(u); GetReg(r)	→	GetReg2(r, u, v)

When operands have selectors, such as indices for arrays, field identifier for records, etc., the path of an operand to a register is longer. Here we show the various state transitions for the three selector kinds index, field designation, and dereferencing.

1. Field designator (x.y, procedure *Field*)

mode	args		result	args	emitted instructions	
Var	a	→	Var	a+y.val	none	(y.val = field offset)
Par	a, b	→	Par	a, b+y.val	none	
RegI	a	→	RegI	a+y.val	none	
RegX	r,a,b	→	RegI	a := y.val	ADD	r', r, a LSL b

The address of a record field is the sum of the record's address and the field offset. Here we encounter the item mode *RegX*. It occurs when both the item's address and the offset are in registers. In this case, the ARM architecture even allows the offset to be shifted, i.e. multiplied by a power of 2. This is most convenient, as the instruction allows to multiply and add in one.

2. Dereferencing (x↑, procedure *Deref*)

mode	args		result	args	emitted instructions	
Var	r, a	→	Par	r, a, b=0	none	
Par	r,a,b	→	RegI	r, a	LDR	r', r, a
RegI	r, a	→	RegI	r, a=0	LDR	r', r, a
RegX	r, a, b	→	RegI	r, a=0	LDR	r', r, a LSL b

The address of the pointer variable is replaced by its value, the address of the dereferenced variable.

3. Index (x[y], procedure *Index*)

mode	args		result	args	emitted instructions
Var	r, a	→	RegI	r, a, b=0	MUL r', r, y.r, r" | ADD r', r, y.r LSL s
RegI	same				r" contains element size previously loaded
RegX	same				y.r = index value previously loaded

The address of an indexed variable is the sum of the array's address and the index times the size of the elements. Its computation therefore requires a multiplication and an addition. Both operations can be combined in a single multiply/add instruction. In the case of several indices, each index contributes a term in the sum. Given an array

a: ARRAY $s_0, s_1, \dots , s_n$ OF T

$$adr(a[i_0, i_1, \ldots , i_n]) \;=\; adr(a) + i_0*n_0 + i_1*s_1 + \ldots + i_n*s_n$$

The multiply/add instruction can be replaced by a single add/shift instruction, if the size *s* is a power of 2. (*Ri LSL n* stands for Ri * 2^n).

Code generated for the various selectors is shown in the following brief example. Note that the instruction pairs *CMP, SWI* serve for checking the validity of index values (array bound checking).

```
MODULE M;
  TYPE
    Record = RECORD u, v, w: INTEGER END ;
    Pointer = POINTER TO Record;
    Array = ARRAY 8 OF INTEGER;
    Matrix = ARRAY 4, 8 OF INTEGER;
  PROCEDURE P;
    VAR i, j, k: INTEGER;
      r: Record; p: Pointer; a: Array; M: Matrix;
  BEGIN k := r.v; k := p^.w; k := a[i]; k := M[i, j]; k := M[3, 6]
  END P;
END M.
```

```
I    -4      adress of variable, based on frame pointer FP
j    -8
k    -12
r    -24
p    -28
a    -60
M    -188
```

```
 7    E51CB014    LDR    R11 FP -20              r.v
 8    E50CB00C    STR    R11 FP -12              k

 9    E51CB01C    LDR    R11 FP -28              p^.w
10    E59BB008    LDR    R11 R11 8
11    E50CB00C    STR    R11 FP -12

12    E51CB004    LDR    R11 FP -4               i
13    E35BBF02    CMP    R11 R11 8
14    2F000001    SWI
15    E08CB10B    ADD    R11 FP R11 LSL  2       a[i]
16    E51BB03C    LDR    R11 R11 -60
17    E50CB00C    STR    R11 FP -12

18    E51CB004    LDR    R11 FP -4               i
19    E35BBF01    CMP    R11 R11 4
20    2F000001    SWI
21    E08CB28B    ADD    R11 FP R11 LSL  5       M[i]
22    E51CA008    LDR    R10 FP -8               j
23    E35AAF02    CMP    R10 R10 8
24    2F000001    SWI
25    E08BB10A    ADD    R11 R11 R10 LSL  2      M[i][j]
26    E51BB0BC    LDR    R11 R11 -188
27    E50CB00C    STR    R11 FP -12

28    E51CB044    LDR    R11, FP, -68            M[3][6]
29    E50CB00C    STR    R11 FP -12
```

Constant expressions are evaluated by the compiler, and so are constant addresses, such as indices with constants. This results in a substantial shortening of code. In order to achieve it, the loading of values must be delayed until it is no longer avoidable.

A rather special case is the indexing of a formal array parameter. Here the address of the element *a[k]* is *adr(a) + k*elsize*, where the address is stored in a parameter location and needs to be loaded into a register. The address being the sum of two registers, it is possible to use the ARM's load or store instruction with 2 register fields rather than a register plus an offset. This facility is the reason for the introduction of the *RegX* item mode. The following example shows the use of this mode, saving an ADD instruction. (The instruction pairs for index checking are omitted).

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

```
  4    E51CB014    LDR   R11 FP -20          k
  8    E51CA004    LDR   R10 FP -4           @x
  9    E51C9014    LDR   R9 FP -20           k
 13    E51C800C    LDR   R8 FP -12           @y
 14    E7989109    LDR   R9 R8 R9 LSL  2
 15    E78A910B    STR   R9 R10 R11 LSL  2
```

9. Integer Expressions and Local Code Improvements

Code generation for all expressions is based on the operand loading principles explained so far. It is straightforward. The RISC architecture requires all operands in registers, and deposits results also in registers. The instructions used are ADD, SUB, and MUL. ADD and SUB come in two forms, namely with the second operand in a register or as an immediate field. This makes compilation significantly more complicated, because (1) the field holds only small values (8 bits), (2) the small constant may be shifted (and become a large constant), and (3) the value is unsigned, hence possibly requiring subtraction in place of addition.

The procedures involved in generating integer arithmetic instructions are:

```
PROCEDURE Neg(VAR x: Item);   (* x := -x *)
PROCEDURE AddOp(op: INTEGER; VAR x, y: Item);   (* x := x ± y *)
PROCEDURE MulOp(VAR x, y: Item);   (* x := x × y *)
PROCEDURE DivOp(op: INTEGER; VAR x, y: Item);   (* x := x div/mod y *)
PROCEDURE Abs(VAR x: Item);   (* x := ABS(x) *)
```

The ARM instruction set does not contain division, because it is a rarely used operation. Hence, division is implemented as a sequence of instruction including a loop. It uses the following algorithm for non-negative operands with a double register for the pair <r, q>:

```
PROCEDURE Div(x, y: INTEGER);
   VAR r, q, i: INTEGER;
BEGIN q := x; r := 0; i := 32;
   REPEAT q := 2*q; r := 2*r;
      IF r >= y THEN r := r-y; q := q+1 END;
      i := i - 1
   UNTIL  i = 0;
   (*q = quotient, r = remainder*)
END Div
```

The statement *z := x DIV y* is translated into the following instructions:

```
  3    E51FB01C    LDR    R11 PC -28       x
  4    E51FA024    LDR    R10 PC -36       y
  5    E35A0000    CMP    R0 R10 0         test for positive divisor
  6    DF000007    SWI
  7    E1B0900B    MOV    R9 R0 R11        q := x
  8    B2799000    RSB    R9 R9 0
  9    E3A08000    MOV    R8 R0 0          r := 0
 10    E3A07020    MOV    R7 R0 32         I := 32
 11    E0999009    ADD    R9 R9 R9         q := q+q
 12    E0B88008    ADC    R8 R8 R8         r := r+r
 13    E158000A    CMP    R0 R8 R10        IF r >= y THEN
 14    A048800A    SUB    R8 R8 R10           r := r - y
 15    A2899001    ADD    R9 R9 1             q := q + 1
 16    E2577001    SUB    R7 R7 1          i := i - 1
 17    1AFFFFF8    BNE       -8
 18    E35B0000    CMP    R0 R11 0         IF x < 0 THEN
 19    AA000003    BGE        3
 20    E2799000    RSB    R9 R9 0             q := -q
 21    E3580000    CMP    R0 R8 0             IF r # 0 THEN
 22    12499001    SUB    R9 R9 1                q := q - 1
 23    104A8008    SUB    R8 R10 R8              r := y - r
```

```
24    E1A0B009    MOV    R11 R0 R9
25    E50FB070    STR    R11 PC -112          z
```

The function ABS(x) is coded in-line, as are all standard functions. This means that no procedure call and return are involved. In this case there are only two instructions involved:

```
ABS(x)          TST   r          (x in register r)
                RSB   r, r, 0    (if negative, reverse subtraction, R[r] := 0 – R[r])
```

The used technique of code generation on-the-fly in conjunction with delayed code emission in the case of constants allows for significant code improvements without much additional complication. (We refrain from using the strong word *optimization*). Apart from the direct evaluation of expressions of constants (constant folding), obvious cases are multiplication and division by a power of 2. Here, a shift is used. The ARM instructions conveniently let the second operand be shifted prior to its addition or subtraction. We have encountered this use already in the address computation for indexed variables.

$x + y * 2^n$	ADD	r, x, y LSL n	(x, y denote registers)
$x - y \text{ DIV } 2^n$	SUB	r, x, y LSR n	

Similarly, the modulus operation is implemented by a masking operation, if the modulus is a power of 2:

$x \text{ MOD } 2^n$	AND	r, x, 2^n-1	if n <= 8
	BIC	r, x, 2^{n-24}-1	if n >= 24
	MOV	r, x LSL 32-n	if 8 < n < 24
	MOV	r, r LSR 32-n	

10. Real Expressions

The ARM-processor – we used DEC's DS1035 in 1996 – does not feature a floating-point unit. Therefore, operations on numbers of type REAL must be programmed using integer arithmetic. We adhere to the IEEE Standard 32-bit format using a sign bit *s*, an 8-bit exponent *e* and a 23-bit mantissa *m* such that $x = (-1)^s \times 2^{e-127} \times 1.m$

First, in the expectation of acquiring a floating-point unit at a later time, when available and if still considered necessary, the author programmed an emulator package. The compiler generated ARM floating-point (FP) instructions that the processor would recognize as undefined coprocessor instructions causing a trap. When it turned out that an FP-unit would neither be forthcoming nor be necessary, I decided to replace the trap interpreter by a set of regular procedures for the basic FP operations. This was an ideal case for using the concept of fast leaf procedure (see below).

```
PROCEDURE Add(x, y: REAL): REAL;
PROCEDURE Multiply(x, y: REAL): REAL;
PROCEDURE Divide(x, y: REAL): REAL;
PROCEDURE Floor(x: REAL): INTEGER;
PROCEDURE Float(x: INTEGER): REAL;
```

Subtraction is performed by addition with a single, preceding instruction inverting the sign bit. Implementation in the form of (leaf) procedures instead of a trap handler improved performance by a factor of 3. This was a big surprise. The primary reason was that saving and restoring the entire bank of registers could be avoided. Furthermore, the quite complicated decoding of instructions into their components turned out to be rather time-consuming too.

A problem arises because according to the calling convention parameters must be deposited in registers R_{11} and R_{10}. This would preclude operators to occur within expressions, as temporary results would be stored in these registers. The usual way out is to push these temporaries onto the stack, and pop them after the operation is executed. An option to avoid this overhead is to provide the routines in several variants, with parameters in registers R_{11} and R_{10}, R_{10} and R_9, R_9 and R_8, etc. respectively. This solution, however, was ultimately deemed unacceptable. Another option is to reserve certain registers for the FP emulator, and to move arguments and results within the register array before and after each call. This solution expands the code inordinately, even for simple cases, and was therefore also rejected. Hence registers are now saved and restored on the stack as shown by the following example. The routines for saving and restoring registers are the same as for procedure calls (see Ch. 16)

```
MODULE M2;
  PROCEDURE P;
    VAR x, y, z, w: REAL;
  BEGIN w := x+y; w := x + (y+z); w := (x–y) * (x+y); w := (x+y) + ((x+y) + (x+y))
  END P;
END M2.
```

```
 4   E51CB004   LDR   R11 FP -4        x
 5   E51CA008   LDR   R10 FP -8        y
 6   EB010000   BL        0            +
 7   E50CB010   STR   R11 FP -16       w

 8   E51CB008   LDR   R11 FP -8
 9   E51CA00C   LDR   R10 FP -12
10   EB010006   BL        6            +
11   E51CA004   LDR   R10 FP -4
12   EB01000A   BL       10            +
13   E50CB010   STR   R11 FP -1

14   E51CB004   LDR   R11 FP -4
15   E51CA008   LDR   R10 FP -8
16   E22AA102   XOR   R10 R10 ...      invert sign
17   EB01000C   BL       12            +
18   E92D0800   STM   SP               push x+y
19   E51CB004   LDR   R11 FP -4
20   E51CA008   LDR   R10 FP -8
21   EB010011   BL       17            +
22   E1A0A00B   MOV   R10 R0 R11
23   E8BD0800   LDM   SP               pop x+y
24   EB020015   BL       21            *
25   E50CB010   STR   R11 FP -16       w

26   E51CB004   LDR   R11 FP -4
27   E51CA008   LDR   R10 FP -8
28   EB010018   BL       24
29   E92D0800   STM   SP               push
30   E51CB004   LDR   R11 FP -4
31   E51CA008   LDR   R10 FP -8
32   EB01001C   BL       28
33   E1A0A00B   MOV   R10 R0 R11
34   E8BD0800   LDM   SP               pop
35   E92D0800   STM   SP               push
36   E1A0B00A   MOV   R11 R0 R10
37   E92D0800   STM   SP               push
38   E51CB004   LDR   R11 FP -4
39   E51CA008   LDR   R10 FP -8
40   EB010020   BL       32
41   E1A0A00B   MOV   R10 R0 R11
42   E8BD0800   LDM   SP               pop
43   EB010028   BL       40
44   E1A0A00B   MOV   R10 R0 R11
45   E8BD0800   LDM   SP               pop
46   EB01002B   BL       43
47   E50CB010   STR   R11 FP -16       w
```

The in-line procedures PACK and UNPK serve to insert and extract the exponent from a floating-point number in an efficient way. PACK(x, e) effectively multiplies x by 2^e. The procedure expects a positive, normalized x, i.e. 1.0 <= x < 2.0. UNPK(x, e) assigns the exponent of *x* to *e* and normalizes *x*.

```
UNPK(x, e)                        PACK(x, e)

LDR   R11 x                       LDR   R11 x
MOV   R10 R11 R11 LSR 23          LDR   R10 e
SUB   R10 R10 127                 ADD   R11 R11 R10 LSL 23
STR   R10 e                       STR   R11 x
```

```
SUB    R11 R11 R10 LSL 23
STR    R11 x
```

11. Set Expressions

Values of type SET are represented by a single word, each bit marking the presence of the corresponding element in the set. For example, the set {1, 3, 6} is represented by the word $00\ldots01001010_2$ (= 4AH), with bits 1, 3, and 6 being ones. As a consequence, set operations are implemented by simple logical operations, namely set union (+) by OR, set intersection (*) by AND, and set difference (-) by BIC. Thus, operations on sets are very efficient.

The singleton set {n} is generated by loading a 1 and then shifting it by *n* bits to the left. The set {0 .. m} is generated by loading the logical complement of 1, i.e. the set {1 .. 31}, shifting *m* bits to the left, yielding {m+1 .. 31}, and complementing again. The set {m .. n} is obtained by the logical subtraction of {0 .. m-1} from {0 .. n}. Only 4 logical instructions are required for this operation. The operation ABS(s), denoting a set's size or number of elements is implemented with a tight counting loop consisting of only 3 instructions. Further details are explained on hand of the following example:

```
MODULE M;
  PROCEDURE P;
    VAR m, n: INTEGER; r, s, t: SET;
  BEGIN s := {}; s := {2, 4, 8 .. 15};
    s := s * t + r; s := s / t - r;
    s := {n}; s := {0 .. m}; s := {m .. n};
    n := ABS(s)
  END P;
END M.

m    -4    (addresses of variables)
n    -8
r    -12
s    -16
t    -20

  4  E3A0B000   MOV   R11 R0 0          {}
  5  E50CB010   STR   R11 FP -16
  6  E59FB080   LDR   R11 PC 128        {2, 4, 8 .. 15}
  7  E50CB010   STR   R11 FP -16
  8  E51CB010   LDR   R11 FP -16        s
  9  E51CA014   LDR   R10 FP -20        t
 10  E00BB00A   AND   R11 R11 R10       *
 11  E51CA00C   LDR   R10 FP -12        r
 12  E18BB00A   OR    R11 R11 R10       +
 13  E50CB010   STR   R11 FP -16
 14  E51CB010   LDR   R11 FP -16
 15  E51CA014   LDR   R10 FP -20
 16  E02BB00A   XOR   R11 R11 R10       /
 17  E51CA00C   LDR   R10 FP -12
 18  E1CBB00A   BIC   R11 R11 R10       -
 19  E50CB010   STR   R11 FP -16

 20  E51CB008   LDR   R11 FP -8         n
 21  E3B0A001   MOV   R10 R0 1
 22  E1B0BB1A   MOV   R11 R0 R10 LSL R11
 23  E50CB010   STR   R11 FP -16        {n}

 24  E51CB004   LDR   R11 FP -4         m
 25  E3E0A001   MVN   R10 R0 1
 26  E1E0BB1A   MVN   R11 R0 R10 LSL R11
 27  E50CB010   STR   R11 FP -16        {0 .. m}

 28  E51CB008   LDR   R11 FP -8         n
 29  E51CA004   LDR   R10 FP -4         m
 30  E3E09001   MVN   R9 R0 1
 31  E1E0BB19   MVN   R11 R0 R9 LSL R11
 32  E3E09000   MVN   R9 R0 0
 33  E00BBA19   AND   R11 R11 R9 LSL R10
 34  E50CB010   STR   R11 FP -16        {m .. n}
```

```
35  E51CB010    LDR    R11 FP -16          s
36  E3A0A000    MOV    R10 R0 0
37  E1B0B0AB    MOV    R11 R0 R11 LSR  1
38  E2AAA000    ADC    R10 R10 0
39  1AFFFFFC    BNE       -4               back to 37
40  E1A0B00A    MOV    R11 R0 R10          ABS(s)
41  E50CB008    STR    R11 FP -8
```

We emphasise that the operations of set construction are remarkably short and fast. This is a testimony to the appropriateness of the ARM instruction set.

The procedures involved in generating set instructions are:

```
PROCEDURE Singleton(VAR x, y: Item);   (* x := {y} *)
PROCEDURE Set(VAR x, y, z: Item);   (* x := {y .. z} *)
PROCEDURE SetOp(op: INTEGER; VAR x, y: Item);   (* x := x op y *)
```

12. Comparisons and Boolean Expressions

Boolean expressions, truth values, most frequently occur as conditions in if-, while-, and repeat statements, whereas Boolean variables occur relatively rarely. Boolean values are typically the results of comparisons. However, compare instructions (CMP) do not deposit a truth value in a register, but rather several truth values in the special register called *condition code*. It typically consists of four bits, indicating whether the result of the operation was zero (Z), negative (N) whether a carry out (C) or an overflow (V) occurred. Most instructions deliver the condition code as a side-effect: In addition to the regular result they yield *four* (not one) additional Boolean values. This arrangement of depositing the result of a comparison in a special register has become standard, and implementations of languages must handle it properly, although it represents a nasty exception of expression evaluation.

Comparisons are straightforward, because they are represented by a single instruction. A new mode, however, must be introduced to represent this special case. It is called the *CC mode*, and if *x.mode = CC, x.r* denotes the condition mask that must be used in conditional branches in order to execute the branch if the specified comparison yields FALSE. The following example illustrates the situation on hand of an if statement. The conditional branch instruction selects a function of the four conditions to decide whether or not the branch is to be taken.

```
E35B0005     CMP    R11 R0 5          IF x = 5 THEN …
1Axxxxxx     BNE    L
```

Boolean expressions with logical operators are handled unlike their arithmetic counterparts. This is mainly because the expression’s value may be known without evaluating the second operand. Its evaluation may therefore be skipped. The language definition even requires it to be skipped, as its evaluation may be impossible. Logical operators are thus represented by conditional branches. This is shown in the following examples, where *x, y*, and *z* stand for comparisons, and BF stands for a conditional branch to be taken only if the corresponding comparison fails, and BT for a branch to be taken if the comparison is satisfied.

```
    x & y & z                      x OR y OR z
    code for x                     code for x
    BF L                           BT L
    code for y                     code for y
    BF L                           BT L
    code for z                     code for z
L: …                           L: …
```

The example shows that (1) the condition (mask) must be known when the branch is generated, and (2) the offset (jump length) of the branch must be filled in (fixed up) when the destination is later known. The first point is solved by letting an item *x* specify the encountered relation by its attribute *x.r* and *x.mode = CC*. This value is generated by procedure *SetCC* and used for the condition field in the branch instruction.

To solve problem (2), the branches with unresolved destinations are chained with links in their offset field. In general, there are two chains, one for BF, the other for BT instructions. Once the destination is known, these chains are traversed and each chain element is replaced by the appropriate destination offset. (See procedure *FixLink*). The item attribute is augmented by the heads of the two chains placed in the fields *x.a* and *x.b*.

Apart from comparisons, Boolean values are also generated by the relations IN and IS. The former is handled by a TST instruction, the latter is discussed in the section on type tests. Furthermore, sets can be compared for inclusion.

```
MODULE M;
    VAR s0, s1: SET; k, n: INTEGER;
BEGIN
    IF s0 <= s1 THEN k := 3 END ;
    IF s0 >= s1 THEN k := 5 END ;
    IF n IN s0 THEN k := 7 END
END M.

 3  E51FB018     LDR    R11 PC -24          s0
 4  E51FA020     LDR    R10 PC -32          s1
 5  E1DBB00A     BIC    R11 R11 R10
 6  1A000001     BNE        1               to 9
 7  E3A0B003     MOV    R11 R0 3
 8  E50FB034     STR    R11 PC -52          k

 9  E51FB030     LDR    R11 PC -48          s0
10  E51FA038     LDR    R10 PC -56          s1
11  E1DAA00B     BIC    R10 R10 R11
12  1A000001     BNE        1               to 15
13  E3A0B005     MOV    R11 R0 5
14  E50FB04C     STR    R11 PC -76

15  E51FB048     LDR    R11 PC -72          s0
16  E51FA058     LDR    R10 PC -88          n
17  E3A09001     MOV    R9 R0 1
18  E11B0A19     TST    R0 R11 R9 LSL R10
19  0A000001     BEQ        1               to 22
20  E3A0B007     MOV    R11 R0 7
21  E50FB068     STR    R11 PC -104
```

The procedures involved in generating conditions are:

```
PROCEDURE IntRelation(op: INTEGER; VAR x, y: Item);        (* x := x < y *)
PROCEDURE SetRelation(op: INTEGER; VAR x, y: Item);        (* x := x < y *)
PROCEDURE RealRelation(op: INTEGER; VAR x, y: Item);       (* x := x < y *)
PROCEDURE CompareArrays(op: INTEGER; VAR x, y: Item);      (* x := x < y *)
PROCEDURE In(VAR x, y: Item);                              (* x := x IN y *)
PROCEDURE Odd(VAR x: Item);                                (* x := ODD(x) *)
PROCEDURETypeTest(VAR x: Item; T: OSAB.Type);              (* x := x IS T *)
PROCEDURE Bit(VAR x, y: Item)
```

13. If, While, and Repeat Statements

The generation of code for conditional and iterative statements follows a simple pattern which is evident through the following examples. References to points in the code (labels) are held in variables of the respective parsing procedures. Thereby care is automatically taken of nested statements through recursion. The same technique of offset fixup is used as for Boolean expressions.

```
PROCEDURE P;
  VAR m, n: INTEGER;
BEGIN
  IF m = 0 THEN n := 33
  ELSIF m < 0 THEN n := 17
  ELSE n := 0
  END ;
  WHILE m > n DO m := m - n  (*gcd*)
  ELSIF n > m DO n := n - m
  END ;
  REPEAT DEC(n) UNTIL n = 0
END P
```

```
 4  E51CB004    LDR   R11 FP -4          m
 5  E35BB000    CMP   R11 R11 0
 6  1A000002    BNE       2              to 10
 7  E3A0B021    MOV   R11 R0 33
 8  E50CB008    STR   R11 FP -8          n
 9  EA000007    BR        7              to 18
10  E51CB004    LDR   R11 FP -4          m
11  E35BB000    CMP   R11 R11 0
12  AA000002    BGE       2              to 16
13  E3A0B011    MOV   R11 R0 17
14  E50CB008    STR   R11 FP -8          n
15  EA000001    BR        1              to 18
16  E3A0B000    MOV   R11 R0 0
17  E50CB008    STR   R11 FP -8          n

18  E51CB004    LDR   R11 FP -4          m
19  E51CA008    LDR   R10 FP -8          n
20  E15B000A    CMP   R0 R11 R10
21  DA000004    BLE       4              to 27
22  E51CB004    LDR   R11 FP -4          m
23  E51CA008    LDR   R10 FP -8          n
24  E05BB00A    SUB   R11 R11 R10
25  E50CB004    STR   R11 FP -4          m
26  EAFFFFF6    BR      -10              to 18
27  E51CB008    LDR   R11 FP -8          n
28  E51CA004    LDR   R10 FP -4          m
29  E15B000A    CMP   R0 R11 R10
30  DA000004    BLE       4              to 36
31  E51CB008    LDR   R11 FP -8          n
32  E51CA004    LDR   R10 FP -4          m
33  E05BB00A    SUB   R11 R11 R10
34  E50CB008    STR   R11 FP -8          n
35  EAFFFFED    BR      -19              to 18

36  E51CB008    LDR   R11 FP -8          n
37  E24BB001    SUB   R11 R11 1
38  E50CB008    STR   R11 FP -8          n
39  E51CB008    LDR   R11 FP -8          n
40  E35BB000    CMP   R11 R11
41 1AFFFFF7     BNE      -7              to 36
```

14. The For Statement

An iteration can be conveniently expressed by a for statement if an integer is progressing over a range of values with a fixed step that can be either positive or negative. The step must be a constant. This is required because it determines the termination condition. The step must be less than 256 in absolute value. This is due to the desire to use an add instruction with immediate operand.

```
PROCEDURE P;
    VAR i, n, k: INTEGER;
BEGIN
    FOR i := 1 TO n BY 2 DO k := k+i END
ENDP
```

```
 4  E3A0B000    MOV   R11 R0 1           1
 5  E51CA008    LDR   R10 FP -8          n
 6  E15B000A    CMP   R0 R11 R10
 7  CA000007    BGT       7              to 16
 8  E50CB004    STR   R11 FP -4          i
 9  E51CB00C    LDR   R11 FP -12         k
10  E51CA004    LDR   R10 FP -4          i
11  E09BB00A    ADD   R11 R11 R10
12  E50CB00C    STR   R11 FP -12         k
13  E51CB004    LDR   R11 FP -4          i
14  E28BB002    ADD   R11 R11 2          +2
15  EAFFFFF4    BR      -12              to 5
```

15. The Case Statement

The case statement serves to select a statement according to an integer, just as an element is selected from an array. This helps to avoid long cascades of if-elsif statements and increases efficiency. However, case statements are recommended only if the set of selectable statements is reasonably large. The compiler generates an internal array of jump offsets corresponding to the component statements. The table entry is chosen by the index specified in the case clause. The resulting code is shown in the following example:

```
PROCEDURE P;
  VAR k: INTEGER;
BEGIN
  CASE k OF
    0: k := 1
  | 1: k := 4
  | 2: k := 16
  | 3: k := 64
  END
END P;
```

```
 4  E51CB004    LDR    R11 FP -4              k
 5  E35B0003    CMP    R0 R11 3               range check
 6  908FB10B    ADD    R11 PC R11 LSL  2
 7  959BB038    LDR    R11 R11 56             Tab[k]
 8  908FF10B    ADD    PC PC R11 LSL  2       switch
 9  EF000004    SWI                           trap, if not in range

10  E3A0B001    MOV    R11 R0 1               case 0
11  E50CB004    STR    R11 FP -4
12  EA00000C    BR        12                  to 26

13  E3A0BF01    MOV    R11 R0 4               case 1
14  E50CB004    STR    R11 FP -4
15  EA000009    BR         9                  to 26

16  E3A0BE01    MOV    R11 R0 16              case 2
17  E50CB004    STR    R11 FP -4
18  EA000006    BR         6                  to 26

19  E3A0BD01    MOV    R11 R0 64              case 3
20  E50CB004    STR    R11 FP -4
21  EA000003    BR         3                  to 26

22  00000000                                  Table of offsets
23  00000003
24  00000006
25  00000009
```

We emphasize that (in contrast to original Oberon) case labels must be integers starting with 0 (like lower array bounds). The table is restricted to 256 entries. Note that there is only a single jump to the selected case, plus one leading to the statement following the cases. Case statements are handled by procedure *Case* in the parser. It contains an array as local variable which holds the entry addresses of the cases indexed by their label. The switch is generated by procedure *CaseHead* in module OSAG, and the offset table is emitted by procedure *CaseTail.*

16. Procedures and Functions

Every procedure invocation establishes an area of storage for its local variables. Local variables and parameters are addressed by adding their static offset to a base address specific to this invocation and held in the register called *frame pointer* (FP).The code for the procedure is headed by a so-called *prolog* that allocates the necessary space in the stack, initializes the frame pointer, and stores the link to the previous frame and the return address. The procedure’s code is terminated by an *epilog* that re-establishes the stack state that existed before the call. Fig. 3 shows the global storage layout and that of two procedure frames on the stack.

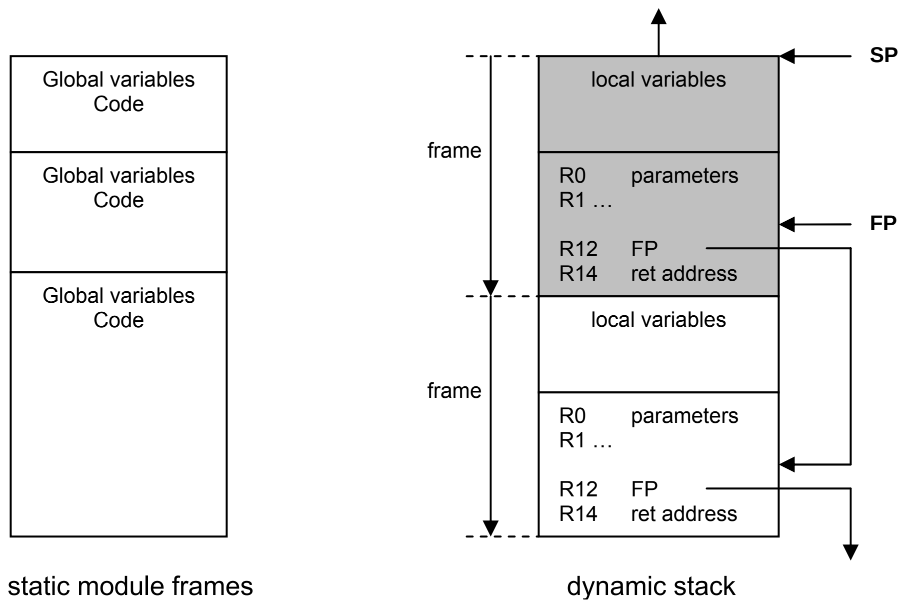


Fig. 3 Storage Layout

The code corresponding to a procedure call consists of a *Branch and Link* instruction, preceded by code evaluating the actual parameters of the call. Value parameters (expressions) are evaluated. In the case of Var-parameters their addresses are generated. Values and addresses are deposited in free registers in descending order, starting with R_{11}. The branch instruction then transfers control to the procedure and deposits the return address in R_{14} (LNK). The prolog then moves the parameters from the registers into memory (except in the case of leaf procedures) and adjusts the base registers FP (frame pointer) and SP (stack pointer). The relevant procedures are *Call* and *SaveRegisters*, the latter being invoked before the compilation of parameters:

```
STM     SP, {parameters, FP, LNK}       push parameters into memory, decrements SP
ADD     FP, SP, m                       FP := SP + size of param block
SUB     SP, SP, n                       SP := SP – size of local variable block
```

SP marks the top of the stack, and FP is the origin of the chain linking the frames. At the bottom of each frame lie the two words holding this link (to the preceding frame) and the call's return address. This pair is establishes by the same STM instruction that stores the parameters in memory.

The epilog restores the stack to the state before the call, and in the case of a function procedure, copies the result onto the top of the (register) stack (by procedure *RestoreRegs*). The last element of the frame chain is then removed, and the return address is moved into the PC. This is again done by a single LDM instruction.

```
MOVR    SP, FP                          SP := FP
LDM     SP, {FP, PC}                    pop from memory FP and PC
```

Prolog and epilog are remarkably short thanks to the convenient load/store multiple instruction of the ARM processor that can be programmed as push and pop operations. As can be seen in the example below, the result of a function is stored in a register (R11), replacing the arguments in registers (R11 , R10, …)

It should be noted that parameters are stored in consecutive registers. This is where they are placed by the routines for expression evaluation. It limits the number of parameters (to at most 12). Of graver consequence is that the register allocation scheme must be such that no gaps are left in the register bank. The register bank must be treated like a stack. Fig. 4 shows the use of the register bank before, during and after a procedure activation.

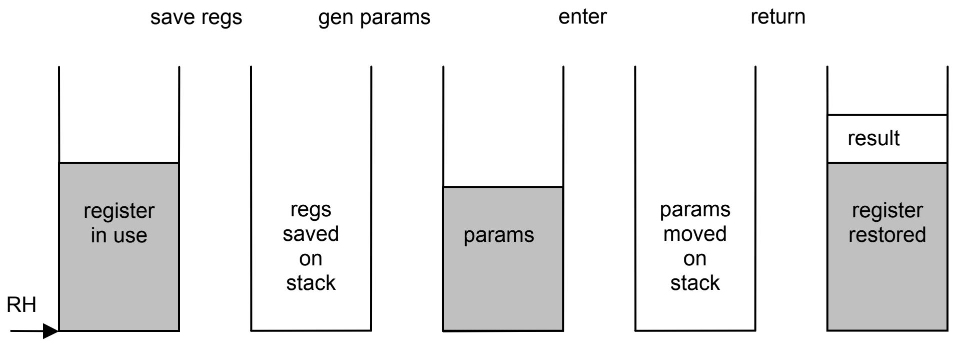


Fig. 4. Register usage on procedure call

```
MODULE M;
  VAR a, b: INTEGER;

  PROCEDURE F(x, y: INTEGER): INTEGER;
    VAR z: INTEGER;
  BEGIN
    IF x > y THEN z := F(x-y, y)
    ELSIF y > x THEN z := F(y-x, x)
    ELSE z := x
    END ;
    RETURN z
  END F;

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

BEGIN a := F(35, 21); Q(a, b)
END M.
```

```
a   -4          variable offsets
b   -8
z  -12
z  -12
```

```
 1   E92D5C00   STM   SP, {R10, R11, FP, LNK}  prolog F
 2   E28DC008   ADD   FP SP 8
 3   E24DD004   SUB   SP SP 4

 4   E51CB004   LDR   R11 FP -4             x
 5   E51CA008   LDR   R10 FP -8             y
 6   E15B000A   CMP   R0 R11 R10
 7   DA000006   BLE       6                 to 15
 8   E51CB004   LDR   R11 FP -4             x
 9   E51CA008   LDR   R10 FP -8             y
10   E05BB00A   SUB   R11 R11 R10
11   E51CA008   LDR   R10 FP -8             y
12   EBFFFFF3   BL       -13                to 1 (recursion)
13   E50CB00C   STR   R11 FP -12
14   EA00000C   BR       12                 to 28
15   E51CB008   LDR   R11 FP -8
16   E51CA004   LDR   R10 FP -4
17   E15B000A   CMP   R0 R11 R10
18   DA000006   BLE       6                 to 26
19   E51CB008   LDR   R11 FP -8
20   E51CA004   LDR   R10 FP -4
21   E05BB00A   SUB   R11 R11 R10
22   E51CA004   LDR   R10 FP -4
23   EBFFFFE8   BL       -24                to 1 (F)
24   E50CB00C   STR   R11 FP -12
25   EA000001   BR        1                 to 28
```

```
26   E51CB004   LDR   R11 FP -4
27   E50CB00C   STR   R11 FP -12

28   E51CB00C   LDR   R11 FP -12         RETURN z
29   E1A0D00C   MOV   SP R0 FP           epilog F
30   E8BD9000   LDM   SP, {FP, PC}

31   E92D5C00   STM   SP, {R10, R11, FP, LNK}  prolog Q
32   E28DC008   ADD   FP SP 8
33   E24DD004   SUB   SP SP 4

34   E51CB004   LDR   R11 FP -4          x
35   E59BB000   LDR   R11 R11 0
36   E50CB00C   STR   R11 FP -12         z
37   E51CB008   LDR   R11 FP -8          y
38   E59BB000   LDR   R11 R11 0
39   E51CA004   LDR   R10 FP -4          x
40   E58AB000   STR   R11 R10 0
41   E51CB00C   LDR   R11 FP -12         z
42   E51CA008   LDR   R10 FP -8          y
43   E58AB000   STR   R11 R10 0

44   E1A0D00C   MOV   SP R0 FP           epilog Q
45   E8BD9000   LDM   SP, {FP, PC}

46   E92D5000   STM   SP, {FP, LNK}      prolog module M
47   E1A0C00D   MOV   FP R0 SP

48   E3A0B023   MOV   R11 R0 35
49   E3A0A015   MOV   R10 R0 21
50   EBFFFFCD   BL       -51             to 1 (F)
51   E50FB0D8   STR   R11 PC -216        a

52   E24FBF37   SUB   R11 PC 220         a
53   E24FAF39   SUB   R10 PC 228         b
54   EBFFFFE7   BL       -25             to 31 (Q)

55   E1A0D00C   MOV   SP R0 FP           epilog M
56   E8BD9000   LDM   S, {FP, PC}
```

A complication arises if a procedure with parameters is called as a variable, and if the variable is indexed or accessed via pointer. The reason lies in the single-pass compilation scheme together with the need to allocate parameters strictly in sequence in registers. This implies that a procedure address cannot lie in a register below the parameters. Our solution is to evaluated the selector and push the address on the stack in order to be popped after the parameters are evaluated. The following example shows, how this case is handled.

```
MODULE M2;
  TYPE PT = PROCEDURE (m, n: INTEGER);
    Object = POINTER TO RECORD w: INTEGER; p: PT END ;
  VAR obj: Object; a: ARRAY 4 OF PT;

  PROCEDURE P(m, n: INTEGER; q: PT; VAR obj: Object);
  BEGIN q(m, n); obj.p(m, n); a[m](m, n)
  END P;
END M2.
```

```
 6  E51CB004   LDR   R11 FP -4          m
 7  E51CA008   LDR   R10 FP -8          n
 8  E51C900C   LDR   R9 FP -12          q
 9  E1A0E00F   MOV   LNK R0 PC
10  E1A0F009   MOV   PC R0 R9           call

11  E51CB010   LDR   R11 FP -16         obj
12  E59BB000   LDR   R11 R11 0          obj↑
13  E59BB004   LDR   R11 R11 4          obj↑.p
14  E52DB004   STR   R11 !SP -4         push branch adr
15  E51CB004   LDR   R11 FP -4          m
16  E51CA008   LDR   R10 FP -8          n
17  E1A0E00F   MOV   LNK R0 PC
18  E4BDF004   LDR   PC !SP 4           pop, call
```

```
19  E51CB004    LDR   R11 FP -4          m
20  E35BBF01    CMP   R11 R11 4          check range
21  2F000001    SWI
22  E08FB10B    ADD   R11 PC R11 LSL 2
23  E51BB074    LDR   R11 R11 -116       a[m]
24  E52DB004    STR   R11 !SP -4         push branch adr
25  E51CB004    LDR   R11 FP -4          m
26  E51CA008    LDR   R10 FP -8          n
27  E1A0E00F    MOV   LNK R0 PC
28  E4BDF004    LDR   PC !SP 4           pop, call
```

17. Records and Pointers

Pointers and records serve to build up complex and dynamic data structures and are a means for defining recursive structures. The components of records are called *fields*, and the compiler assigns fixed offsets relative to the record's origin. Records are either static variables, or they are accessed indirectly via pointers. The value of a pointer variable is the address of the referenced record, or it is NIL, when no record is referenced. A record is dynamically allocated by the intrinsic procedure NEW that is contained in the service module MAU (memory allocation unit). The compiler automatically generates an import entry in the object file, if MAU is referenced.

```
MODULE M2;
  TYPE Rec = RECORD u, v, w: INTEGER END ;
    Ptr = POINTER TO Rec;

  PROCEDURE Q;
    VAR p, q: Ptr;
  BEGIN NEW(p); p.w := 9; q := NIL
  END Q;

END M2.
```

```
 7  E24CBF01    SUB   R11 FP 4           adr of p
 8  E24FAF09    SUB   R10 PC 36          adr of TD
 9  EB010000    BL        0              call to NEW

10  E3A0B009    MOV   R11 R0 9
11  E51CA004    LDR   R10 FP -4          p
12  E58AB008    STR   R11 R10 8          p.w

13  E3A0B000    MOV   R11 R0 0           NIL
14  E50CB008    STR   R11 FP -8          q
```

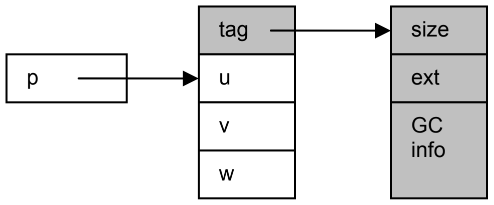


Fig. 5. Record allocation and pointer assignment

A call of NEW(p) allocates a record and assigns its address to the pointer variable *p*. This implies that every pointer is bound to a record type. This binding is fixed with the pointer's declaration. For every record type a *type descriptor* (TD) is created. It is used for recording the hierarchy of extensions of a type (see below), and contains the necessary meta information about the type that is needed by a garbage collector. The latter information is not generated in this current implementation, with the exception of the record's size. Every record so created is prefixed with a field pointing to its type descriptor. It is called the *tag*. Fig. 5 shows the effect of a call to NEW.

Pointer type declarations are in so far an anomaly, as they allow forward references. It is permissible that the base type (a record) of a pointer type be unknown when scanning the text sequentially. This exception from the general rule that everything being referenced must have been previously declared, causes more difficulties than might be expected. Our solution is to record all pointer type declarations in a list rooted in the global variable *pbsList*. Its elements of type *PtrBase* contain the name of the (possibly not yet defined) base type and a pointer to the pointer declaration. When a record type declaration is encountered, this list is scanned, and the base type in the pointer type descriptor is updated. At the end of a declaration section, the list is scanned again for still undefined base types (indicated by the integer type).

18. Assignment of Arrays

Assignments to arrays are also expressed by the regular assignment operator. The ":=" symbol looks rather inconspicuous, considering that copying entire arrays may be a very heavy operation. In contrast to original Oberon, where a the standard procedure COPY is used, the ":=" symbol now also applies to arrays and strings.

Therefore it is mandatory to implement array assignments in an efficient way using in-line code. In this case, the ARM's post-increment addressing mode appears as most useful. Here, the absolute address is in a register, and the instruction's offset field specifies the amount by which this address is incremented (or decremented) after the memory access. This increment is typically 4 (1 word). The copying operation is then executed by a tight loop of only 4 instructions (at 6 – 9 in the following code excerpt):

```
TYPE A = ARRAY 100 OF INTEGER;
VAR dst, src: A;
    dst := src
 3  E51CB004   LDR   R11 FP -4      adr of dst
 4  E51CA008   LDR   R10 FP -8      adr of src
 5  E3A09064   MOV   R9 R0 100      length
 6  E4BA8004   LDR   R8 R10 4       load word from src
 7  E4AB8004   STR   R8 R11 4       store word to dst
 8  E2599001   SUB   R9 R9 1        decrement count
 9  1AFFFFFB   BNE      -5          to 6
```

The arrays must have the same element type, but the source array may be shorter than the destination array. Hence, the length (count) is taken from the source. The length comparison is performed by the compiler. This is not possible in the case of open arrays, when the check must be performed at run-time. (Currently, only the source may be an open array). See procedures *OSAP.StatementSequence* and *OSAG.CopyArray*.

```
  PROCEDURE Copy(VAR dst: A; VAR src: ARRAY OF INTEGER);
  BEGIN dst := src
  END Copy;

25  E51CB004   LDR   R11 FP -4      adr of dst
26  E51CA008   LDR   R10 FP -8      adr of src
27  E51C900C   LDR   R9 FP -12      length of src
28  E3590064   CMP   R0 R9 100      compare with length of dst
29  CF000003   SWI                  trap
30  E4BA8004   LDR   R8 R10 4       load word from src
31  E4AB8004   STR   R8 R11 4       store word to dst
32  E2599001   SUB   R9 R9 1        decrement count
33  1AFFFFFB   BNE      -5          to 30
```

The ARM hardware is word-oriented. Therefore, copying is always done word by word, rather than byte by byte. As a consequence, the length of character (and Boolean) arrays is always rounded up to the next multiple of 4. Still, simple variables and record fields of these types use a single byte only.

19. Dynamic Arrays

Dynamic arrays are here introduced as an addition to Oberon. An array is called *dynamic*, if its length is determined "dynamically", i.e. at execution time. This is done by a call to the intrinsic procedure NEW with a second parameter indicating the desired length. Example:

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

```
BEGIN … NEW(a, len) …
```

where *len* is an expression of type INTEGER and a non-negative value. The mechanism for open arrays (parameters) is reused. The array is accessed indirectly via a descriptor consisting of two words. The word with the higher address contains the array's address, the other its length. In the symbol table, dynamic arrays are characterized by their length being -1, their size 8.

```
PROCEDURE P;
  VAR k: INTEGER;
    a: ARRAY OF INTEGER;
BEGIN NEW(a, 10); a[k] := 100
END P;

  4   E24CBF02   SUB    R11 FP 8            adr a
  5   E3A0A00A   MOV    R10 R0 10           len
  6   E3A09F01   MOV    R9 R0 4             element size
  7   EB030000   BL          0              NEW (updated by loader)

  8   E51CB004   LDR    R11 FP -4           k
  9   E51CA00C   LDR    R10 FP -12          len a
 10   E15B000A   CMP    R0 R11 R10          index check
 11   2F000001   SWI
 12   E51CA008   LDR    R10 FP -8           a[k]
 13   E3A09F19   MOV    R9 R0 100
 14   E78A910B   STR    R9 R10 R11 LSL  2
```

Dynamic arrays cannot be elements of other data structures.

20. Strings

Strings are sequences of characters. As they are classified as constants, they represent an exceptional case in so far as they are structured constants, the only ones in the language. Constants are typically allocated as immediate values in the instructions. But this is impossible for structured constants requiring individual amounts of storage. They can neither be allocated in a single word, nor can they be stored as a literal in an immediate field of an instruction. Indeed, they present somewhat of a problem to the compiler designer, as they do not fit into the regular scheme of constant handling. Strings are loaded into the buffer *strings* (from OSAS.string) by procedure *MakeStringItem,* when the scanner delivers the symbol *string.*.

Hence the question arises where to allocate strings. The only possibility is to place them after the program code. As the address is unknown when the instruction is issued, the string is stored temporarily in a table of string constants (TSC). At the end of each procedure, the table is scanned and the necessary address fixups are made (OSAG.FixupConstants). Notably, the same technique is used for other constants which do not fit into the very short immediate field of ARM instructions, and for indirect addresses for imported variables (tables TIC, TXR). As the address field of instructions is also very short, the offsets are rather limited. It has turned out that the updating of addresses and the flushing of the tables, cannot always wait until the end of the currently compiled procedure is reached, as the offsets would become too large. Therefore, the need for flushing the tables is checked at the end of every statement. In this case, however, a jump instruction must be issued to cross over the inserted constants. This is a most ugly feature, but the ARM's instruction coding unfortunately makes it necessary.

Declared string constants do not require fixups, as they are put into the code array when declared. Therefore their address is known. Both string literals and declared strings are constants, but are referenced like variables (mode = Var). The trick of treating them like variables in the code generator has the advantage, that also for export/import the necessary mechanisms are already available and can be readily used. In reality, however, initially export of variables was not permitted, and the mechanism had to be introduce to handle string constants (and type descriptors), and then could of course also be used for variables.

Assignments of strings to variables is identical to that of arrays, namely by copying word for word in a tight loop. This is efficient, and it is possible, because of the postulate that string lengths and the lengths of character arrays be always a multiple of 4. However, termination is not due to a count reaching zero, but by detecting the string's terminating null character.

In contrast to general arrays, character arrays and strings can be compared, as they are (alphabetically) ordered values. Comparison proceeds character by character in a tight loop. Repetition terminates, if either an unequal pair or a zero character is encountered. The details can be seen from the following brief example:

```
MODULE M;
  CONST S = "abcdefg";
  VAR k: INTEGER;
    a, b: ARRAY 16 OF CHAR;
BEGIN a := S; b := "ABCDE";
  IF a = b THEN k := 1 END ;
  IF a < "012" THEN k := 3 END ;
END M.
```

```
k  -4           addresses of variables
a  -20
b  -36

 1  64636261     abcdefg   S
 2  00676665

 5  E24FBE03     SUB    R11 PC 48           a
 6  E24FAF07     SUB    R10 PC 28           S
 7  E4BA9004     LDR    R9 R10 4
 8  E4AB9004     STR    R9 R11 4
 9  E21994FF     AND    R9 R9 0FF000000
10  1AFFFFFB     BNE      -5                to 7

11  E24FBF16     SUB    R11 PC 88           b
12  E28FAF1B     ADD    R10 PC 108          @41
13  E4BA9004     LDR    R9 R10 4
14  E4AB9004     STR    R9 R11 4
15  E21994FF     AND    R9 R9 0FF000000
16  1AFFFFFB     BNE      -5                to 13

17  E24FBE06     SUB    R11 PC 96           a
18  E24FAF1D     SUB    R10 PC 116          b
19  E4FB9001     LDRB   R9 R11 1
20  E4FA8001     LDRB   R8 R10 1
21  E1590008     CMP    R0 R9 R8
22  1A000001     BNE       1                to 25 if unequal
23  E3590000     CMP    R0 R9 0             0X ?
24  1AFFFFF9     BNE      -7                to 19
25  1A000001     BNE       1                to 28
26  E3A0B001     MOV    R11 R0 1
27  E50FB078     STR    R11 PC -120         k := 1

28  E24FBF23     SUB    R11 PC 140          a
29  E28FAE03     ADD    R10 PC 48           @43
30  E4FB9001     LDRB   R9 R11 1
31  E4FA8001     LDRB   R8 R10 1
32  E1590008     CMP    R0 R9 R8
33  1A000001     BNE       1                to 36 if unequal
34  E3590000     CMP    R0 R9 0             0X ?
35  1AFFFFF9     BNE      -7                to 30
36  AA000001     BGE       1                to 39 if greater or equal
37  E3A0B003     MOV    R11 R0 3
38  E50FB0A4     STR    R11 PC -164         k := 3

41  44434241     ABCDE
42  00000045
43  00323130     012
```

The relevant compilation procedures are *Declarations* in *OSAP,* and *AllocString, EnterString, FixupConstants, MakeItem, CompareArrays, CopyString,* and *StringParam* in OSAG.

21. Type Extension, Type Tests and –Guards

Static typing is an important principle in programming languages. It implies that every constant, variable or function is of a certain data type, and that this type can be derived by reading the program text without executing it. It is the key principle to introduce important redundancy in languages in such a form that a compiler can detect inconsistencies. It is therefore the key element for reducing the number of errors in programs.

However, it also acts as a restriction. It is, for example, impossible to construct data structures (arrays, trees) with different types of elements. In order to relax the rule of strictly static typing, the notion of *type extension* was introduced in Oberon. It makes it possible to construct inhomogeneous data structures without abandoning type safety. The price is that the checking of type consistency must in certain instances be deferred to run-time. Such checks are called *type tests*. The challenge is to defer to run-time as few checks as possible and as many as needed.

The solution in Oberon is to introduce families of types, and compatibility among their members. Their members are thus related, and a family forms a hierarchy. The principle idea is the following: Any record type T0 can be extended into a new type T1 by additional record fields (attributes). T1 is then called an *extension* of T0, which in turn is said to be T1's *base type*. T1 is then type compatible with T0, but not vice-versa. This property ensures that in many cases static type checking is still possible. Furthermore, it turns out that run-time tests can be made very efficient, thus minimizing the overhead for maintaining type safety.

For example, given the declarations

```
TYPE R0 = RECORD u, v: INTEGER END ;
     R1 = RECORD (R0) w: INTEGER END
```

we say that R1 is an extension of R0. R0 has the fields u and v, R1 has u, v, and w. The concept becomes useful in combination with pointers. Let

```
TYPE P0 = POINTER TO R0;
     P1 = POINTER TO R1;
VAR p0: P0;  p1: P1;
```

Now it is possible to assign p1 to p0 (because a P1 is always also a P0), but not p0 to p1, because a P0 need not be a P1. This has the simple consequence that a variable of type P0 may well point to an extension of R0. Therefore, data structures can be declared with a base type, say P0, as common element type, but in fact they can individually differ, they can be any extension of the base type.

Obviously, it must be possible to determine the actual, current type of an element even if the base type is statically fixed. This is possible through a *type test*, syntactically a Boolean factor:

```
p0 IS P1                    (short for p0^ IS R1)
```

Furthermore, we introduce the *type guard*. In the present example, the designator p0.w is illegal, because there is no field *w* in a record of type R0, even if the current value of p0^ is a R1. As this case occurs frequently, we introduce the short notation *p0(P1).w*, implying a test *p0 IS P1* and an abort if the test is not met.

It is important to mention that this technique also applies to formal variable parameters of record type, as they also represent a pointer to the actual parameter. Its type may be any extension of the type specified for the formal parameter in the procedure heading.

How are type test and type guard efficiently implemented? Our first observation is that they must consist of a single comparison only, similar to index checks. This in turn implies that types must be identified by a single word. The solution lies in using the unique address of the type descriptor of the (record) type. Which data must this descriptor hold? Essentially, type descriptors (TDs) must identify the base types of a given type. Consider the following hierarchy:

```
TYPE T =   RECORD … END ;
     T0 =  RECORD (T) … END ;         extension level 1
     T1 =  RECORD (T) … END ;         extension level 1
     T00 = RECORD (T0) … END ;        extension level 2
     T01 = RECORD (T0) … END ;        extension level 2
     T10 = RECORD (T1) … END ;        extension level 2
     T11 = RECORD (T1) … END ;        extension level 2
```

In the symbol table, the field *base* refers to the ancestor of a given record type. Thus base of the type representing T11 points to T1, etc. Run-time checks, however, must be fast, and hence cannot proceed through chains of pointers. Instead, each TD is an array with references to the ancestor TDs (including itself). For the example above, the TDs are as follows:

```
TD(T)  =     [T]
TD(T0) =     [T, T0]
TD(T1) =     [T, T1]
TD(T00) =    [T, T0, T00]
TD(T01) =    [T, T0, T01]
TD(T10) =    [T, T1, T10]
TD(T11) =    [T, T1, T11]
```

Evidently, the first element can be omitted, as it always refers to the common base of the type hierarchy. The last element always points to the TD's owner. Like constants, TDs are allocated in the code area.

References to TDs are called *type tags*. They are required in two cases. The first is for records referenced by pointers. Such dynamically allocated records carry an additional, hidden field holding their type tag. The second case is that of record-typed VAR-parameters. In this case the type tag is explicitly passed along with the address of the actual parameter. Such parameters therefore require two words/registers.

A type test then consists of a test for equality of two type tags. In *p IS T* the first tag is that of the n'th entry of the TD of *p^*, where n is the extension level of T. The second tag is that of type *T*. This is shown in the following example.

```
MODULE M;
  TYPE R0 = RECORD x: INTEGER END ;
       R1 = RECORD (R0) y: INTEGER END ;
       R2 = RECORD (R1) z: INTEGER END ;
       P0 = POINTER TO R0;
       P1 = POINTER TO R1;
       P2 = POINTER TO R2;
  VAR k: INTEGER;
       p0: P0; p1: P1; p2: P2;
BEGIN
  IF p0 IS P2 THEN k := 3 END ;
  k := p0(P2).z
END M.

k  2   -4
p0  2   -8
p1  2  -12
p2  2  -16

  1-3   TD(R0)
  4-6   TD(R1)
  7-9   TD(R2)

 12    E51FB040    LDR    R11 PC -64      p0
 13    E51BA004    LDR    R10 R11 -4      tag(p0^)
 14    E59AA008    LDR    R10 R10 8       TD(p0^)[2]
 15    E24F9F0A    SUB    R9 PC 40        tag(R2)
 16    E15A0009    CMP    R0 R10 R9
 17    1A000001    BNE        1           to 20, if test not met
 18    E3A0B003    MOV    R11 R0 3
 19    E50FB058    STR    R11 PC -88      k

 20    E51FB060    LDR    R11 PC -96      p0
 21    E51BA004    LDR    R10 R11 -4      tag(p0^)
 22    E59AA008    LDR    R10 R10 8       TD(p0^)[2]
 23    E24F9F12    SUB    R9 PC 72        tag(R2)
 24    E15A0009    CMP    R0 R10 R9
 25    1F000002    SWI                    trap, if test not met
 26    E59BB008    LDR    R11 R11 8       p0.z
 27    E50FB078    STR    R11 PC -120     k
```

When declaring a record type, it is not known how many extensions, nor how many levels will be built on this type. Therefore TD's should actually be infinite arrays. We decided to restrict them to 2 levels only. The first entry of the arrays with 3 elements currently contains the size of the record. If a garbage collector will be available, the size may be replaced by a pointer to an extended type descriptor holding further meta-data about the type.

22. Leaf procedures and Register Variables

It is one of the requirements of real-time applications that selected parts of a program be coded most efficiently, i.e. that code is "optimized". This requirement can hardly be met by a simple and fast compiler. On the other hand, also predictability and perspicuity are requirements. And they are best met by straight-forward code generators. Sequences of instructions must not be a riddle to the programmer who decides to inspect the generated code. Small changes in a program must not cause large changes in the code.

Here we thought a satisfactory compromise would lie a simple compiler strategy at large, together with facilities for code improvement through explicit programming hints at selected places, such as real-time critical sections. Three facilities came to mind, the first being register variables, the second leaf procedures, and the third array riders.

Register variables are those that are specified to remain allocated in registers rather than being stored in memory. If such variables are to be determined by the compiler, the resulting algorithm is rather complex, requiring much time and space. The alternative is to let the programmer explicitly specify such variables. Of course, registers are a most scarce resource; hence only very few such variables can exist, and they must be carefully selected. Also, there lurks the danger that temporary relocation to memory may be necessary, e.g. in the case of procedure calls, thus reducing the benefit, and sometimes even converting it into a burden.

On every call, the frame pointer FP and the return address are pushed onto the stack, in addition to all parameters. In the case of procedures that do not call other procedures this effort could be spared, resulting in faster calls and returns. We call such procedures *leaf procedures*, as they are the leaf nodes in the tree structure of calls. This property could be detected by the compiler, requiring, however, a multi-pass strategy, complicating the entire compiler. Again, we opt for a facility to let the programmer specify that a procedure is a leaf.

In Oberon-SA, the facilities of register variables and leaf procedures are combined. A leaf procedure is specified by an asterisk following the symbol PROCEDURE. It then acquires the following properties and restrictions:

1. It cannot contain any calls (except of in-line coded, standard procedures).
2. Parameters are left in registers (R11, R10, …) and are not stored on the stack.
3. Its local variables of type INTEGER or SET are allocated in registers (R11, R10, …)

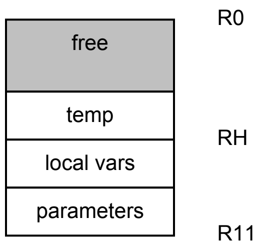


Fig. 6. Register usage in leaf procedures

The measure by which leaf procedures can reduce code length is evident from the following short example. The code for four simple assignments is shown, first with normal compilation, and then with register variables. The number of instructions is reduced from 18 to 7; all memory accesses are eliminated.

```
PROCEDURE P(x, y: INTEGER; VAR z: INTEGER);
   VAR w: INTEGER;
BEGIN w := 10; w := x+y; w := x DIV 16; z := (x+y)*(x-y)
END P;
```

```
 4   E3A0B00A    MOV    R11 R0 10
 5   E50CB010    STR    R11 FP -16          w
 6   E51CB004    LDR    R11 FP -4           x
 7   E51CA008    LDR    R10 FP -8           y
 8   E09BB00A    ADD    R11 R11 R10         x+y
 9   E50CB010    STR    R11 FP -16          w
10   E51CB004    LDR    R11 FP -4           x
11   E1B0B24B    MOV    R11 R0 R11 ASR  4
12   E50CB010    STR    R11 FP -16          w

13   E51CB004    LDR    R11 FP -4           x
14   E51CA008    LDR    R10 FP -8           y
15   E09BB00A    ADD    R11 R11 R10
16   E51CA004    LDR    R10 FP -4           x
17   E51C9008    LDR    R9 FP -8            y
18   E05AA009    SUB    R10 R10 R9
19   E01B0B9A    MUL    R11 R0 R11 R10
20   E51CA00C    LDR    R10 FP -12          z
21   E58AB000    STR    R11 R10 0
```

Code when P is compiled as leaf procedure:

```
 3   E3A0000A    MOV    R0 R0 10            w := 10
 4   E09B000A    ADD    R0 R11 R10          w := x+y
 5   E1B0024B    MOV    R0 R0 R11 ASR  4    w := x DIV 16
 6   E09B800A    ADD    R8 R11 R10          t0 := x+y
 7   E05B700A    SUB    R7 R11 R10          t1 := x-y
 8   E0180897    MUL    R8 R0 R8 R7         t0 := t0 * t1
 9   E5898000    STR    R8 R9 0             z := t0
```

Leaf procedures and register variables seem to be innocent enough facilities to be implemented without hesitation. However, register variables are nastier to implement than expected. For example, if an addition is issued, the result typically replaces the arguments in the registers. But this cannot be in the case of arguments being register variables, because generally the variable's values are not to be changed, and instead a temporary register must be allocated. If the store operation in turn stores into a register (variable), then the former assignment to a temporary register may be replaced. In fact, only the destination field in the add instruction need be changed. This sort of fixup is added to the *OSAG.Store* procedure, and is needed only for register variables. Another place where they require a special treatment is the for statement. Here we show the code for the example of Sect. 14 when part of a leaf procedure. The number of instructions in the loop is reduced from 10 to 4:

```
PROCEDURE* P;
     VAR i, n, k: INTEGER;
BEGIN
     FOR i := 1 TO n BY 2 DO k := k+i END
END P
```

```
 3   E3A00000    MOV    R0 R0 1             i := 1
 4   E1500001    CMP    R0 R0 R1
 5   CA000002    BGT        2               to 9
 6   E0922000    ADD    R2 R2 R0            k := k+i
 7   E2800002    ADD    R0 R0 2             i := i+2
 8   EAFFFFFA    BR        -6               to 4
```

The third facility of *array riders* had been implemented on an earlier version of the Oberon-SA compiler, and it resulted in significant speedup for sequential access with arrays. However, we recognized that this situation occurs relatively rarely, whereas the compiler was made more complex for good. Consequently, we decided to drop that feature and will not further discuss it here.

23. Interrupt Procedures

Interrupt handlers are simply parameterless procedures which are not invoked by regular procedure calls, but instead by interrupt signals breaking the normal instruction fetch sequence. Although the ARM switches to different PC and SP registers upon an interrupt, that is, effectively switches to a different stack, we need not bother about this. The only place where unfortunately code needs to be different from that of regular procedures is the return instruction. In this case, it must move not only the return address from the stack into the PC, but also reload the PSR. An offset value must be provided that is specific to each interrupt source. In Oberon-SA, this offset must be explicitly specified in the procedure declaration. The following example shows a rudimentary handler for the ARM's IRQ interrupt. It merely increments a global counter.

```
MODULE M;
  IMPORT SYSTEM;

  CONST IntVec = 18H; StkOrg = 800H;
  VAR count: INTEGER;

  PROCEDURE Handle [4];
  BEGIN INC(count)
  END Handle;

BEGIN count := 0;
  SYSTEM.PUT(18H, (SYSTEM.ADR(Handle) - IntVec - 8) DIV 4 + 0EA000000H);
  SYSTEM.LDPSR(0, 0D2H); SYSTEM.SP := StkOrg; (*set stack pointer*)
  SYSTEM.LDPSR(0, 53H);  (*enable IRQ interrupt*)
  .....
END M.
```

```
 1  E92D5FFF   STM   SP                save registers on stack
 2  E28DC030   ADD   FP SP 48          FP := SP + 12*4
 3  E51FB018   LDR   R11 PC -24        INC(count)
 4  E28BB001   ADD   R11 R11 1
 5  E50FB020   STR   R11 PC -32
 6  E24CD030   SUB   SP FP 48          SP := FP – 12*4
 7  E8BD5FFF   LDM   SP                restore registers
 8  E25EF004   SUB   PC LNK 4          special return (with offset = 4)

11  E3A0B000   MOV   R11 R0 0          entry of module body
12  E50FB03C   STR   R11 PC -60        count := 0

13  E24FBF0E   SUB   R11 PC 56
14  E25BBF06   SUB   R11 R11 24
15  E25BBF02   SUB   R11 R11 8
16  E1B0B14B   MOV   R11 R0 R11 ASR  2
17  E59FA02C   LDR   R10 PC 44
18  E05BB00A   SUB   R11 R11 R10
19  E3A0AF06   MOV   R10 R0 24
20  E58AB000   STR   R11 R10 0         store at 18H

21  E3A0B0D2   MOV   R11 R0 210
22  E129F00B   MSR   PC R9 R11         load program status register
23  E3A0DB02   MOV   SP R0 2048
24  E3A0B053   MOV   R11 R0 83
25  E129F00B   MSR   PC R9 R11         load program status register
```

The assignment of a branch instruction pointing to the handler to the interrupt location at address 18H is performed using a PUT operator (from pseudo module SYSTEM)

24. Import and Export

From the point of view of system design, the module is the most essential structure. It defines not only a set of data types, variables and procedures, but it carries an *interface*, a restricted view of the module visible to other modules, so-called *clients*. The interface describes the set of all *exported* objects. In Oberon, objects to be exported are marked in their declaration by an asterisk (export mark). Modules cannot be nested. They must be considered as units in a universe, and they can *import* objects exported by other, *server* modules. The imported modules are listed in the heading of each module.

Modules cannot only be programmed in relative isolation from other modules, but they can also be compiled separately. However, if a module is compiled, the interfaces of all modules in its import list must be available in order to check for type consistency. This interface is provided in the form of a *symbol file*. The compiler generates a code file to be read by the loader, and a symbol file to be read by the compiler when compiling client modules.

Therefore, the various modules of a system are compiled *separately*, but *not independently*. It is a cornerstone of safe system design that these interfaces are encoded rather than available as sections of source text that can be freely manipulated. In order to avoid version conflicts, code and symbol files carry a key which is checked whenever a symbol file is accessed. The procedures for generating and for reading a symbol file lie in module OSAB.

24.1. Export, Generating a symbol file

Generating a symbol file essentially consists of the linearisation of the data structure describing objects. The data structure, a list of *Objects*, is scanned, and marked objects are written to the file. Hence, a symbol file is an excerpt of the symbol table containing only the descriptions of exported objects. This excerpt is read by the importing compilation and added to its symbol table.

Objects have types, and types are referenced by pointers. These cannot be written on a file. The straight-forward solution would be to use the type identifiers as they appear in the program to denote types. However, this would be rather crude and inefficient, and second, there are anonymous types, for which artificial identifiers would have to be generated.

An elegant solution lies in consecutively numbering types. Whenever a type is encountered the first time, it is assigned a unique reference number. For this purpose, records of type *Type* contain the field *ref*. Following the number, a description of the type is then written to the symbol file. When the type is encountered again during the traversal of the data structure, only the reference number is issued. The global variable *Ref* in OSAB functions as the running reference number.

A symbol file must not contain addresses (of variables or procedures). If they did, most changes in the program would result in a change of the symbol file. This must be avoided, because changes in the implementation (rather than the interface) of a module are supposed to remain invisible to the clients. Only changes in the interface are allowed to effect changes in the symbol file, requiring recompilation of all clients. Therefore, addresses are replaced by *export numbers*. The variable *expno* (local to *OSAB.Export*) serves as running number. The translation from export number to address is performed by the loader. Every code file contains a list (table) of addresses (entry points for procedures). The export number serves as index in this table to obtain the requested address.

Export numbers are generated while writing the symbol file. They are not stored in the objects in the symbol table. However, it must be guaranteed that objects have the same numbers in the symbol file and in the code file. This “synchronization” is a somewhat subtle problem, and it might have simplified matters if the numbers would be stored in the objects explicitly.

The syntax of symbol files follows from the syntax of the data structure of declared objects, i.e. from the syntax of declarations.

```
SymFile =    key name versionkey {object}.
object  =    (CON name type form (value | expno) | TYP type [{fix} 0] | VAR name type expno).
type  =      ref [name [modname key]] ( PTR type | ARR type len | REC type {field} 0 | PROC type {param} 0].
field =      FLD name type offset.
param =      (CON | VAR | PAR) type.
```

If a type’s reference number r is negative, then –r designates the type which had already been output. A zero reference number signals an anonymous type. Procedures are considered to be constants of a procedural type, which is characterized by the types and number of parameters, and in the case of a function, by its result type. The names of formal parameters need not be listed, as they are of no relevance except inside the procedure itself. The “value” of a procedure is its export number. Note that the syntax of *object* specifies a name for constants and variables, but not for types. This anomaly, as well as the sequence of *fixes*, will be explained later.

The following example of a module with exported constants, types, variables, and procedures shows the resulting symbol file in an appropriately decoded form. *#r* precedes the specification of the type with reference number *r*. *^r* is simply a reference to that type. (^3 refers to the standard type CHAR, ^4 to INTEGER, ^6 to SET, ^9 to no type).

```
MODULE A;
  CONST Ten* = 10; Dollar* = "$";
  TYPE R* = RECORD u*: INTEGER; v*: SET END ;
    S* = RECORD w*: ARRAY 4 OF R END ;
    P* = POINTER TO R;
    A* = ARRAY 8 OF INTEGER;
    B* = ARRAY 4, 5 OF REAL;
    C* = ARRAY 10 OF S;
    D* = ARRAY OF CHAR;
  VAR x*: INTEGER;
  PROCEDURE Q0*;
  BEGIN END Q0;
  PROCEDURE Q1*(x, y: INTEGER): INTEGER;
  BEGIN RETURN x+y END Q1;
END A.
```

```
A 1CA7F2A6
 CON Ten [^4]   10
 CON Dollar [^3]   36
 TYP [#14 R  form = REC [^9]  lev = 0  size = 8 { v [^6]  off = 4 u [^4]  off = 0}]
 TYP [#15 S  form = REC [^9]  lev = 0  size = 32 { w [  form = 12 [^14]  len = 4  size = 32]  off = 0}]
 TYP [#16 P  form = PTR [^14]]
 TYP [#17 A  form = ARR [^4]  len = 8  size = 32]
 TYP [#18 B  form = ARR [  form = ARR [^5]  len = 5  size = 20]  len = 4  size = 80]
 TYP [#19 C  form = ARR [^15]  len = 10  size = 320]
 TYP [#20 D  form = ARR [^3]  len = -1  size = 8]
 VAR x [^4]    1
 CON Q0 [  form = PROC [^9] ()]    2
 CON Q1 [  form = PROC [^4] ( class = 2 [^4] class = 2 [^4])]    3
```

24.2. Import, Reading a symbol file

Importing a module causes a new module object to be inserted in the set of global objects. The reading of a symbol file is then guided by the syntax given above, and it causes the construction of a list of the read objects, anchored in the corresponding module object. The parser is based on the principle of recursive descent.

The process of translating reference numbers back to pointers uses an array of pointers to records of type *Type*. When a positive number *r* is read, the subsequent type description is read and results in a data structure. The pointer to it is assigned to *typdesc[r]*. This array is declared globally in OSAB. (It could in fact be declared local to procedure *Import*). In later readings of reference numbers (with negative values), the reference is taken from *typdesc[r]*.

This looks all reasonably straight-forward. However, there is one circumstance that complicates the program considerably. It is the fact that imported types can be re-exported. Consider the example:

```
MODULE A;
  TYPE T* = RECORD x: INTEGER END ;
END A.

MODULE B;
  IMPORT A;
  PROCEDURE P*(VAR t: A.T);
  END P;
END M01.
```

```
MODULE C;
   IMPORT B;
   VAR r: A.T;
BEGIN B.P(r)
END C.
```

Module *C* imports type *T* directly from *A*, but also indirectly from *B* through the parameter *t* of procedure *P*. When compiling *C* it is necessary to determine that the formal parameter of *B.P* and the actual parameter *r* are indeed of the same type. A crude solution would be to let the symbol file of B contain that of A. However, this would lead to the undesirable fact that typically a symbol file would contain all files of all modules of a whole system. The solution rather lies in duplicating only the description of the very type that is to be re-exported. However, when exported, this type need be attached to its name. It is therefore mandatory to export the type's name together with the type's description. (This is the reason for the syntactic anomaly mentioned above: The name is attached to the *type* rather than the *object* in the symbol file). The decoded symbol file of module B is shown below:

```
CON P [form = PROC [^9]( class = PAR [#14 A.T 0A004B5C  form = REC [^9]  lev = 0  size = 4 {}])]    1
```

When a (named) type description is read, a search determines whether the same module is already present. If it is present, the type object is searched in the existing list. If the given type is present, the new input is discarded. This is important in order to recognize that two imports are of the same type. Type equality is determined through the equality of the pointers referring to the compared type descriptors.

In fact, it is possible that the import of a module remains completely hidden. Consider the example where an implicit reference to a type *T* in a module *A* occurs in two modules, say *B1* and *B2*, and where both modules are imported by yet another module *C*. The compatibility, in fact the identity, of the types of variables *p0* and *p1* can be established thanks to their types referring to *A*.

```
MODULE B1;
   IMPORT A;
   TYPE T* = PROCEDURE (VAR u: A.T);
END B1.

MODULE B2;
   IMPORT A;
   TYPE T* = PROCEDURE (VAR u: A.T);
END M03.

MODULE C;
   IMPORT B1, B2; (*twice hidden import of A.T*)
   VAR p0: B1.T;  p1: B2.T;
BEGIN p0 := p1
END C.
```

The symbol file of B1 is

```
TYP [#14 T  form = PROC [^9]( class = PAR [#15 A.T 0600C360  form = REC [^9]  lev = 0  size = 4 { x [^4]  off = 0}])]
```

If a type is imported again and then discarded, it is mandatory that this occurs before a reference to it is established elsewhere. This implies that types must always be defined before they are referenced. Fortunately, this requirement is fulfilled by the language and in particular by the one-pass strategy of the compiler. However, there is one exception, namely the possibility of forward referencing a record type in a pointer declaration, allowing for recursive data structures:

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

Hence, this case must be treated in an exceptional way, i.e. the definition of P must not cause the inclusion of the definition of R, but rather cause a forward reference in the symbol file. Such references must by fixed up when the pertinent record declaration had been read. This is the reason for the term *{fix}* in the syntax of (record) types. Furthermore, the recursive definition

```
TYPE P = POINTER TO RECORD x, y: P END
```

suggests that the test for re-import must occur before the type is established, i.e. that the type's name must precede the type's description in the symbol file, where the arrow marks the fixup.:

```
TYP [#14 P  form = PTR [^1]]
TYP [#15 R  form = REC [^9]  lev = 0  size = 8 { y [^14]  off = 4 x [^14]  off = 0}]  → 14
```

Modules may be imported and given a different name (alias) from the one they have in the environment. For example,

```
IMPORT M0 := M1
```

imports module *M1* and associates the local name *M0* with it, i.e. the identifier *M0* is used as usual, but the file with name *M1* is read.

From these brief explanations is should be evident that the implementation of the module concept together with that of separate compilation is the most complex and most sophisticated part of the entire compiler. Its proper solution was by no means obvious, and it even caused an extensive redesign of the handling of import and export, including a new definition of the format of symbol files after a considerable period of usage.

25. Loading and Linking

Code files contain the compiled instruction sequences. At the beginning stand the module name (a string terminated by 0X) and the module key (4 bytes). They are followed by triples, one per imported module, consisting of the imported modules' name, key, and a fixup anchor. This sequence of triples is terminated by a zero byte. Then follows the sequence of commands, the sequence of entry points, and finally the code. The syntax of code files is:

```
CodeFile  =  modname key
      {importname key anchor} 0
      {commandname entry} 0
      nofentries {entry}
      datasize codesize {code}.
```

After allocating memory in sequential fashion, the linker finalizes the addresses of instructions referring to imported objects. For every import, the instructions pointing to an object of this module are linked by a chain, whose anchor was read together with the individual modules' names and keys. The links of this (fixup) chain are located in the address fields of the linked instructions.

There are two kinds of external references. The first and more frequent one is for procedures. In their 24-bit address (offset) field lies the link (16 bits) and the export number (8 bits) of the called procedure. The latter is used as index in the entry address table read at the beginning of the file. The actual offset value is obtained from this entry offset and the difference of the base addresses of the referencing and the referenced module.

The second kind of external reference is for variables, string constants, and type descriptors. In this case, access is always indirectly. The compiler allocates a word in the area for constants, into which the linker places the direct absolute address of the object to be referenced. Like for procedures, it is obtained from the entry offset and the difference of the module bases.

The two kinds of elements of the chain are distinguished through the word's "opcode" byte, which is a BL in the case of references to procedures, and zero in the case of variables and type descriptors.

This simple strategy of linking and loading is very efficient. It rests on the premise that a system maintains a chain of loaded module descriptors, to which new modules are added on demand. It guarantees that no module occurs more than once. The three page program makes use of recursion.

26. Miscellaneous Topics

In this chapter we add some comments on various features that seem to need exceptional treatment, because they do not fit well into a simple, regular structure, yet are found to be indispensable, convenient, or merely conventional. This compiler implementor could very well have done without them.

26.1. Record and Pointer Types

Oberon clearly distinguishes between records and pointers to records, between direct and indirect access. This is of great value for the conceptual clarity so important in teaching. It requires that there are separate declarations for record and pointer types, as shown in the subsequent example:

```
TYPE R = RECORD a, b: INTEGER END ;
    P = POINTER TO R
```

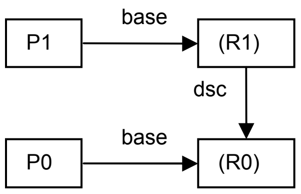
This distinction is also reflected in the denotation of pointers and referenced records: If *p* denotes a pointer variable, then *p* stands for its pointer value, whereas *p^* stands for the dereferenced record and *p^.b* for its field *b*. As this combination of dereferencing and field selection occurs very frequently, the Oberon rules allow the abbreviation *p.b*. This adds a single statement to the compiler, but the programmer must keep in mind that it includes an abbreviation.

In the majority of cases, however, a single type identifier in place of *P* and *R* would indeed suffice, for example if all instantiations of records are dynamic, and hence all records are referenced via pointers. In this case, Oberon allows to let the record type remain anonymous:

```
TYPE P = POINTER TO RECORD a, b: INTEGER END
```

This looks all very well. But in the case of a record extension, it is the record and not the pointer which is extended, and therefore a name for the record type is necessary. Again, the rules of Oberon help with an abbrevaition. It is, for example, possible to specify, using *P0* instead of *R0*

```
TYPE P0 = POINTER TO RECORD a, b: INTEGER END
TYPE P1 = POINTER TO RECORD (P0) c, d: INTEGER END
```


$$\begin{tikzcd}
\boxed{P1} \arrow[r, "base"] & \boxed{(R1)} \arrow[d, "dsc"'] \\
\boxed{P0} \arrow[r, "base"] & \boxed{(R0)}
\end{tikzcd}$$

The two abbreviations described in the preceding section are convenient, and fortunately do not add complexity to the compiler in a noticeable degree. A few statements suffice. Nevertheless, they represent an exception in the regular structure of the languages and the compiler, the sort of things that should be avoided for obtaining a regular design.

26.2. Type descriptors

It was another subtle case that for some time caused difficulties: Typically, a record type needs to have its own identity. Any dynamic allocation scheme requires that types have a descriptor (at run-time) that indicates the type's size. If a garbage collector is to be present, then much more (meta) information is needed, such as the offsets of all pointers. Another instance requiring the availability of type descriptors (TDs) is the concept of type extension. The descriptors must display the hierarchy of the related types. The identity of each type is established by the unique address of its descriptor.

For a long time we believed that record types not connected with pointers (dynamic allocation) nor extensions (type hierarchies) could do without TDs, and that the compiler should not allocate a superfluous TD in these cases. The chosen solution was to always associate a TD with a record type, but to use during compilation the object record associated with the type for holding the TD's address. Hence, the record of type *Object* not only contains the name of the type, but also identifies its descriptor as a variable. This was given up later, recognizing that the descriptor conceptually belongs to the type rather than the (named) object. The descriptor itself is allocated in the heading of the code section, and descriptors are always treated as global variables.

26.3. Forward references

Another feature falling into the same class is that of forward references. In Oberon they exist for two kinds of objects. The first is for procedures. It is required if procedures refer to each other. In simple cases such as

```
PROCEDURE P(x: INTEGER);
BEGIN … Q(x-1) …  (*forward reference*)
END P;

PROCEDURE Q(x: INTEGER);
BEGIN … P(x-1) …
END Q;
```

nesting will solve the problem:

```
PROCEDURE P(x: INTEGER);
    PROCEDURE Q(x: INTEGER);
    BEGIN … P(x-1) …
    END Q;
BEGIN … Q(x-1) …
END P;
```

In more complicated situations, the introduction of a procedure variable will provide the solution, although it may appear as heavy. In any case, we have decided not to implement a forward declaration facility for the current time, as it is rarely needed and awkward to implement in a single-pass compiler.

The second instance of a forward reference is for pointer types, where recursive relations are frequent, as shown in the following example:

```
TYPE T = POINTER TO RECORD x: INTEGER; y: T END
```

However, the declaration

```
TYPE T = RECORD x: INTEGER; y: T END
```

must be detected as an error. Of greater relevance is the case with mutual reference, such as

```
TYPE P0 = POINTER TO RECORD a: P0; b: P1 END ;
    P1 = POINTER TO RECORD a: P0; b: P1 END ;
```

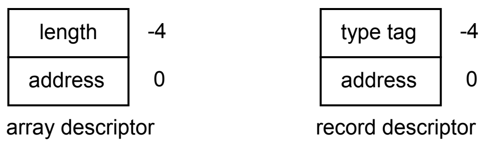
Here a forward declaration is the only possible remedy:

```
TYPE P0 = POINTER TO R0;  (*forward*)
  P1 = POINTER TO R1;  (*forward*)
  R0 = RECORD a: P0; b: P1 END ;
  R1 = RECORD a: P0; b: P1 END ;
```

A solution for its implementation is not difficult to find. Yet, it is an exceptional facility cutting through regular structures, and not without the possibility of unforeseen consequences. We use the form *NoTyp* to stand for the still unknown base of a pointer type, which is later replaced when the actual base type is declared. This implies that the necessity of updating (fixing up) the missing base entry must be recognized. In order to avoid undue complication, we require that a pointer and its associated record type must be defined within the same scope. Therefore, undefined pointer bases can be retained in a table, which at the end of the current declaration section is scanned. An error is indicated, if the table remains not empty. Details are to be looked up in the source text (procedure *OSAP.Declarations*).

26.4. Open Arrays

Open arrays require a treatment different from other arrays only for index bound checking. This is because the bound is not known. It is not a constant, but a variable. Therefore, the length of the array parameter is passed along with the array address, occupying a second register. This represents an exception of the pleasant rule that every procedure parameter occupies a single register, just as in the case of record parameters. Since registers are allocated from higher to lower numbers, and because it is possible to generate the array length only after parsing the designator, the array length is placed in the register below the address, and similarly for the array parameter stored in memory. The only other case, where a parameter occupies two words is the record, where the second word contains the address of the record's type descriptor.



26.5. Export of variables

The export of variables is a questionable practice. The important principle of information hiding demands that only constants be exported. Types and procedures are also considered to be constants. Variables inside a module will be manipulated indirectly by calling exported procedures rather than by direct access. This principle allows to postulate invariants for the module, and to guard them effectively.

Both Modula and Oberon allow to export variables, thus counteracting against a sound principle of software design. Oberon-2 allowed the specification that a variable would be exported in read-only mode, thus making it a constant in the module's environment, and an earlier version of OberonSA strictly forbade the export of variables. We decided to stick to this rule, but to allow for exceptions. We do allow the export of variables of basic, scalar types and of pointers, in read-only mode. This measure avoids the necessity of trivial function procedures for merely inspecting the variable's value. The export of structured variables is not permitted.

26.6. Coping with syntactic errors

Coping with syntactically ill-formed text is an art in itself. There are no fixed rules; heuristics govern the subject. The programmer, however, expects helpful error diagnostics. A basic requirement is that the compiler does not crash, that it "survives" any misformed text. A good strategy to meet this requirement is to design the parser first and fill in the necessary additions for error handling. This guarantees that incorrect syntax is at least discovered, even if not optimally diagnosed. The temptation to modify the basic parser in order to accommodate "frequent programming mistakes" is large, but it easily leads to certain syntax errors being accepted without complaint.

The advice to leave the basic parser untouched also applies to the handling of other errors, in particular type errors. If, for example, a decision can be taken either on the ground of syntax (what is the next symbol?) or on type information, then the former is recommended. There are, in Oberon, a few instances, however, where type information is used for parsing. For example, a statement starting with an identifier is an assignment, if the identifier denotes a variable, or a call, if it denotes a procedure. Fortunately, however, these cases are very few.

There is no limit in spending efforts for elegant error recovery, and therefore the temptation to go far in order to please the programmer is big. The effect is that the entire compiler becomes bulky and slow. In fact it is surprising how large a part of the compiler goes into coping with errors, be they syntactic or about type mismatch. I have never shown extreme sympathy with programmers who do not cope with simple language rules. Sophisticated error recovery and diagnostics will merely honor their carelessness, and keep them from learning the rules. The amount of effort a compiler designer invests in this subject is largely a matter of taste and opinion. Only wisdom imposes limits.

26.7. Run-time traps

Similar considerations can be made regarding run-time errors. Many experts say that a language establishes a set of abstractions and rules governing them, and that any violation of them must be detected and reported. This is a heavy burden, and good languages let this task be handled by the compiler, so that it complicates “only” the compiler. However, some limitations imposed by the abstraction can only be detected when executing the program. The cases at hand are:

1. array index out of bounds
2. type guard violation
3. length of destination array too small for assignment
4. case value out of defined range
5.
6. string too long for assignment
7. division by zero

In early languages, there were no such checks at run-time. But in the meantime programmers have slowly learned to appreciate the usefulness of such checks. Yet, they cost efficiency, and therefore have been questioned. Indeed, they are useless and detrimental to the perfect program. Therefore it was suggested that there should be a compiler switch for turning the generation of run-time checks on or off. They would be turned on as long as a program is tested, and turned off thereafter. But when does a program leave its testing phase? And it is after the testing phase, when the program is “in the field”, when errors become most dangerous and costly.

In any case, the code for checks must be as short and unobtrusive as possible. This compiler meets this requirement admirably well, as can be seen from the following examples. The availability of a conditional trap instruction SWI is most helpful:

```
MODULE M;
  TYPE R0 = RECORD x: INTEGER END ;
    R1 = RECORD (R0) y: INTEGER END ;
    P1 = POINTER TO R1;
  VAR i, k: INTEGER;
    p: POINTER TO R0;
    A: ARRAY 8 OF INTEGER;

  PROCEDURE P(VAR a: ARRAY OF INTEGER);
    VAR b: ARRAY 8 OF INTEGER;
  BEGIN b := a
  END P;

BEGIN
  CASE k: 2 OF
   | 0: k := A[i];
   | 1: k := p(P1).x;
  END
END M.
```

```
 10  E24CBF0A   SUB   R11 FP 40          b := a
 11  E51CA004   LDR   R10 FP -4
 12  E51C9008   LDR   R9 FP -8
 13  E3590008   CMP   R0 R9 8            LEN(a) >= LEN(b)
 14  CF000003   SWI                      array length check

 23  E51FB06C   LDR   R11 PC -108
 24  E35B0003   CMP   R0 R11 2           0 <= k < 2
 25  308FF10B   ADD   PC PC R11 LSL  2   branch indexed
 26  EF000004   SWI                      case check

 30  E51FB084   LDR   R11 PC -132
 31  E35BBF02   CMP   R11 R11 8          0 <= i < 8
 32  2F000001   SWI                      index check

 37  E51FB0A8   LDR   R11 PC -168        p
 38  E51BA004   LDR   R10 R11 -4
 39  E59AA004   LDR   R10 R10 4
 40  E24F9F26   SUB   R9 PC 152          TD(P1)
 41  E15A0009   CMP   R0 R10 R9
 42  1F000002   SWI                      type guard
```

A detail is worth while being pointed out. Because in Oberon the lower bound of array indices is fixed at zero, a single comparison will suffice to test both lower and upper bound, if the index is interpreted as an *unsigned* integer.

26.8. “Optimizations”

The terms *optimization* and *to optimiz*e are frequently abused in the field of computing. *Improvement* and *to improve* would be more honest terms. To reach for the optimum is often unwise and expensive. However, we will here bow to tradition and use this established term.

Where an effort should be made to improve code, that is, where to improve the compiler’s code generators, is an open question. It depends largely on the features of the target computer’s instruction set. It may also feature instructions to improve efficiency. In general, one should investigate the frequency with which a feature is used before embarking on an optimization trip. RISC architectures usually do not offer many such instances, as in principle the instructions set is supposed to be minimal and can rarely be further shortened.

We have adopted one simple rule for the choice of optimizations: Restrict optimizations to cases, where the programmer has no means to avoid the construct. But we will not bother to improve code, when the programmer could do this by programming differently. For example, this compiler does not “optimize” in the following cases:

```
x := x + 0          x := x * 1          x := x
x = x               x < x
ASSERT(TRUE)        ODD(0)
```

However, the compiler evaluates expressions consisting of integer constants only. This is necessary, because such expressions may occur in declarations, for example, in specifying an array length. Care has to be taken against overflow, as the compiler must never crash. This is also the reason why we do not bother about constant expression evaluation in the case of type REAL.

Noteworthy because highly effective is the use of shift instructions for multiplication and division by integer constants that are powers of 2. Obviously, the compiler needs to check for such exceptional cases for every multiplication and division. The use of shifts for multiplication by a constant is particularly beneficial in combination with register variables, as the following example shows:

```
MODULE M;
  PROCEDURE* P;
    VAR i, k: INTEGER;
  BEGIN k := 2*i; k := 3*i;
    k := 9*i + i;  (* 10*i *)
    k := i*5 * 2  (* 10*i *)
  END P;
END M.
```

```
 3   E1B01080    MOV    R1 R0 R0 LSL  1           k := 2*i
 4   E0901080    ADD    R1 R0 R0 LSL  1           k := 3*i
 5   E090B180    ADD    R11 R0 R0 LSL  3
 6   E09B1000    ADD    R1 R11 R0                 k := 8*I + i  (= 10*i)
 7   E090B100    ADD    R11 R0 R0 LSL  2
 8   E1B0108B    MOV    R1 R0 R11 LSL  1          k := 2*(4*I + i)  (= 10*i)
```

But most important are optimizations in address generation, where the programmer has no means to improve code. Such improvements are relatively easy to implement, because they are local. The same holds for assignment of the null string. Instead of s := “”, the statement s[0] := 0X is emitted. Hence, the code for a loop is avoided.

We end this section about optimization with a few examples, where the code appears as “suboptimal”, but where we do not bother to improve it, knowing that these cases occur rarely. It concerns BOOLEAN expressions and assignments, which are a special case defying the conventional rules of expression evaluation, and involving the special register called *condition code*. In the case of variables, the truth values are represented by 0 and 1, stored in a single byte, in fact by its least bit. As soon as BOOLEAN operators are involved, the condition code register becomes involved:

```
p := p & q
```

```
  16    E55FB04A     LDRB   R11 PC -74           q
  17    E35B0000     TST    R0 R11 1             test bit 0 only
  18    0A000002     BEQ        2                to 22
  19    E55FB055     LDRB   R11 PC -85           p
  20    E35B0000     TST    R0 R11 1
  21    1A000001     BNE        1                to 24
  22    E3A0B000     MOV    R11 R0 0
  23    EA000000     BR         2                to 25
  24    E3A0B001     MOV    R11 R0 1             TRUE
  25    E54FB06D     STRB   R11 PC -109          p
```

Evidently, this simple assignment results in a rather cumbersome code sequence. The lesson is to use Boolean variables sparingly. A shortcut is used by the compiler, however, if a simple variable (or a constant) is negated. It is a rare case where we make use of the ARM's facility of conditional execution of single instructions.

```
p := ~q; p := ~TRUE;

   7    E55FB026     LDRB   R11 PC -38           q
   8    E35B0000     TST    R0 R11 1
   9    03A0B001     MOV    R11 R0 1
  10    13A0B000     MOV    R11 R0 0
  11    E54FB035     STRB   R11 PC -53           p

  12    E1500000     CMP    R0 R0 R0             TRUE
  13    13A0B001     MOV    R11 R0 1
  14    03A0B000     MOV    R11 R0 0
  15    E54FB045     STRB   R11 PC -69           p
```

This last example looks queer and is, of course, a pathological case. Because the ARM does not allow for a condition FALSE (never), it (and with it also its negation) has to be generated "artificially". This is done by comparing a register with itself. This is, of course, not the only instance where a compiler must cope with pathological constructs!

27. Interface Definitions

```
DEFINITION OSAS;  (*the Scanner*)
  CONST IdLen* = 32;
    (*lexical symbols*)
    times* = 1; rdiv* = 2; div* = 3; mod* = 4;
    and* = 5; plus* = 6; minus* = 7; or* = 8; eql* = 9;
    neq* = 10; lss* = 11; leq* = 12; gtr* = 13; geq* = 14;
    in* = 15; is* = 16; arrow* = 17; period* = 18;
    char* = 20; int* = 21; real* = 22; false* = 23; true* = 24;
    nil* = 25; string* = 26; not* = 27; lparen* = 28; lbrak* = 29;
    lbrace* = 30; ident* = 31; if* = 32; case* = 33; while* = 34;
    repeat* = 35; for* = 36;
    comma* = 40; colon* = 41; becomes* = 42; upto* = 43; rparen* = 44;
    rbrak* = 45; rbrace* = 46; then* = 47; of* = 48; do* = 49;
    to* = 50; by* = 51; semicolon* = 52; end* = 53; bar* = 54;
    else* = 55; elsif* = 56; until* = 57; return* = 58;
    array* = 60; record* = 61; pointer* = 62; const* = 63; type* = 64;

  TYPE Ident* = ARRAY IdLen OF CHAR;
  VAR ival*, slen*: LONGINT;  (*results of Get*)
    rval*: REAL;
    id*: Ident;  (*for identifiers*)
    str*: ARRAY 60 OF CHAR;  (*for strings*)
    errcnt*: INTEGER;

  PROCEDURE CopyId*(VAR ident: Ident);
  PROCEDURE Mark*(msg: ARRAY OF CHAR);
  PROCEDURE Get*(VAR sym: INTEGER);
  PROCEDURE Init*(T: Texts.Text; pos: LONGINT);
END OSAS.


DEFINITION OSAB;  (*basis of data type definitions*)
  CONST versionkey = -1;
    (* class values*) Head = 0;
```

```
        Const = 1; Var = 2; Par = 3; Fld = 4; Ty* = 5;
        SProc = 6; Mod = 7; Reg = 8; RegI = 9;

      (* form values*)
        Byte = 1; Bool = 2; Char = 3; Int = 4; Real = 5; Set = 6;
        Pointer = 7; NilTyp = 8; NoTyp = 9; Proc = 10;
        String = 11; Array = 12; Record = 13;

    TYPE Object = POINTER TO ObjDesc;
      Type = POINTER TO TypeDesc;

      ObjDesc= RECORD
        class, lev: INTEGER;
        expo, rdo: BOOLEAN;   (*exported, read-only*)
        next, anc: Object;
        type: Type;
        name: OSAS.Ident;
        val: LONGINT
      END ;

      TypeDesc = RECORD
        form, ref: INTEGER;  (*ref is only used for import/export*)
        nofpar: INTEGER;  (*for procedures, extension level for records*)
        len: LONGINT;  (*for arrays, len < 0 => open*)
        dsc, typobj*: Object;
        base: Type;  (*for arrays, records, pointers*)
        size: LONGINT  (*in bytes; always multiple of 4, except for Bool and Char*)
      END ;

    VAR topScope, guard: Object;
      byteType, boolType, charType: Type;
      intType, realType, setType, nilType, noType, strType: Type;

    PROCEDURE NewObj(): Object; (*insert new Object with name OSAS.id*)
    PROCEDURE this(): Object;  (*return the Object with name OSAS.id*)
    PROCEDURE thisimport(mod: Object): Object;
    PROCEDURE thisfield(rec: Type): Object;
    PROCEDURE OpenScope;
    PROCEDURE CloseScope;
    PROCEDURE MakeFileName(VAR name, FName: OSAS.Ident; ext: ARRAY OF CHAR);

    PROCEDURE Import(VAR modid, modid1: OSAS.Ident);
    PROCEDURE Export(VAR modid: OSAS.Ident;
  END OSAB.
```

```
  DEFINITION OSAG;   (*code genertor*)
    CONST WordSize = 4; FP = 12; MaxCase = 256;

    PROCEDURE FixLink(L0: LONGINT);
    PROCEDURE AllocString(VAR adr: LONGINT);
    PROCEDURE AllocTD(tp: OSAB.Type);
    PROCEDURE Val(VAR x: Item): LONGINT;
    PROCEDURE Lev(VAR x: Item): LONGINT;
    PROCEDURE MakeConstItem(VAR x: Item; typ: OSAB.Type; val: LONGINT);
    PROCEDURE MakeRealItem(VAR x: Item; val: REAL);
    PROCEDURE MakeStringItem(VAR x: Item; VAR str: ARRAY OF CHAR);
    PROCEDURE MakeItem(VAR x: Item; y: OSAB.Object);

    PROCEDURE Field(VAR x: Item; y: OSAB.Object);   (* x := x.y *)
    PROCEDURE Index(VAR x, y: Item; check: BOOLEAN);   (* x := x[y] *)
    PROCEDURE DeRef(VAR x: Item);
    PROCEDURE TypeTest(VAR x: Item; T: OSAB.Type; varpar, isguard: INTEGER);

    PROCEDURE And1(VAR x: Item);   (* x := x & *)
    PROCEDURE And2(VAR x, y: Item);
    PROCEDURE Or1(VAR x: Item);   (* x := x OR *)
    PROCEDURE Or2(VAR x, y: Item);
    PROCEDURE Neg(VAR x: Item);   (* x := -x *)
    PROCEDURE AddOp(op: INTEGER; VAR x, y: Item);   (* x := x op y *)
    PROCEDURE MulOp(VAR x, y: Item);   (* x := x * y *)
    PROCEDURE DivOp(op: INTEGER; VAR x, y: Item);   (* x := x op y *)
    PROCEDURE PrepOpd(VAR x: Item; op: INTEGER; VAR rst: SET);
    PROCEDURE RealOp(op: INTEGER; VAR x, y: Item; rst: SET);   (* x := x op y *)
```

```
   PROCEDURE Singleton(VAR x, y: Item);  (* x := {y} *)
   PROCEDURE Set(VAR x, y, z: Item);   (* x := {y .. z} *)
   PROCEDURE In(VAR x, y: Item);  (* x := x IN y *)
   PROCEDURE SetO(op: INTEGER; VAR x, y: Item);   (* x := x op y *)

   PROCEDURE IntRelation(op: INTEGER; VAR x, y: Item);   (* x := x < y *)
   PROCEDURE SetRelation(op: INTEGER; VAR x, y: Item);   (* x := x < y *)
   PROCEDURE RealRelation(op: INTEGER; VAR x, y: Item);   (* x := x < y *)
   PROCEDURE StringRelation(op: INTEGER; VAR x, y: Item);   (* x := x < y *)

   PROCEDURE PrepStore(VAR x: Item);
   PROCEDURE Store(VAR x, y: Item); (* x := y *)
   PROCEDURE CopyRecord(VAR x, y: Item);  (* x := y *)
   PROCEDURE CopyArray(VAR x, y: Item);  (* x := y *)
   PROCEDURE CopyString(VAR x, y: Item);  (* x := y *)

   PROCEDURE VarParam(VAR x: Item; ftype: OSAB.Type);
   PROCEDURE ValueParam(VAR x: Item);
   PROCEDURE StringParam(VAR x: Item);
   PROCEDURE ByteParam(VAR x: Item);  (*formal param of type SYSTEM.BYTES*)

   PROCEDURE CaseHead(VAR x: Item; VAR L0, L1: LONGINT);
   PROCEDURE CaseTail(VAR ctab: ARRAY OF LONGINT; L0, L1, max: LONGINT);
   PROCEDURE For0(VAR x, y: Item);
   PROCEDURE For1(VAR x, y, z, w: Item; VAR L: LONGINT);
   PROCEDURE For2(VAR x, y, w: Item);

   PROCEDURE Here(VAR L: LONGINT);
   PROCEDURE FJump(VAR L: LONGINT);
   PROCEDURE CFJump(VAR x: Item);
   PROCEDURE BJump(L: LONGINT);
   PROCEDURE CBJump(VAR x: Item; L: LONGINT);
   PROCEDURE Fixup(VAR x: Item);
   PROCEDURE PrepCall(VAR x: Item; VAR rst: SET);
   PROCEDURE Call(VAR x: Item; rst: SET);
   PROCEDURE Header;
   PROCEDURE Enter(leaf, int: BOOLEAN; level: INTEGER; regvarno, parsize, varsize: LONGINT);
   PROCEDURE Return(leaf, int: BOOLEAN; offset, form: INTEGER; VAR x: Item);

   PROCEDURE Increment(upordown: LONGINT; VAR x, y: Item);
   PROCEDURE Assert(VAR x, y: Item);
   PROCEDURE New(VAR x, y: Item);
   PROCEDURE Pack(VAR x, y: Item);
   PROCEDURE Unpk(VAR x, y: Item);
   PROCEDURE Get(VAR x, y: Item);
   PROCEDURE Put(VAR x, y: Item);
   PROCEDURE PSR(op: LONGINT; VAR msk, x: Item);  (*Program Status Reister*)
   PROCEDURE CPR(op: LONGINT; VAR cpno, cpreg, x: Item);  (*Coprocessor Register*)
   PROCEDURE Flush(VAR x: Item);  (*flush caches*)
   PROCEDURE AddC(VAR x, y, z: Item);
   PROCEDURE MulD(VAR x, y, z: Item);

   PROCEDURE Abs(VAR x: Item);
   PROCEDURE Odd(VAR x: Item);
   PROCEDURE Floor(VAR x: Item; rst: SET);
   PROCEDURE Float(VAR x: Item; rst: SET);
   PROCEDURE Ord(VAR x: Item);
   PROCEDURE Len(VAR x: Item);
   PROCEDURE Shift(fct: LONGINT; VAR x, y: Item);
   PROCEDURE Adr(VAR x: Item);
   PROCEDURE Bit(VAR x, y: Item);
   PROCEDURE Xor(VAR x, y: Item);
   PROCEDURE Overflow(VAR x: Item);
   PROCEDURE Null(VAR x: Item);    (*must be register variable*)

   PROCEDURE CheckRegs;
   PROCEDURE Open;
   PROCEDURE Close(VAR modid: OSAS.Ident; key, datasize: LONGINT);
END OSAG.
```

```
DEFINITION OSAP;   (*parser*)
   PROCEDURE Compile;
END OSAP.
```

References

N. Wirth and J. Gutknecht. *Project Oberon.* Addison-Wesley, 1992. ISBN 0-201-54428-8
N. Wirth. *Compiler Construction.* Addison-Wesley, 1996. ISBN 0-201-40353-6
- *Grundlagen und Techiken des Compilerbaus.* Addison-Wesley, 1996. ISBN 3-98319-931-4