Eidgenössische
Technische
Hochschule
Zürich

*Institut
für
Informatik*

Urs Ammann

*On Code
Generation
in a PASCAL
Compiler*

On Code Generation in a PASCAL Compiler

by

Urs Ammann

Abstract: PASCAL has been implemented at the Swiss Federal Institute of Technology in Zurich for the CDC 6000 computer series. This report deals with code generation in the resulting one-pass compiler (which is written in the language it compiles). The paper gives insight into the runtime organization of data as well as into the use of the hardware registers. It is shown how the compiler maintains a description of the register contents and uses this description to generate efficient code. Several examples of compiled code are discussed.

Author's address: Institut für Informatik
                  ETH-Zentrum
                  CH-8092 Zürich

Table of Contents

1. Introduction

PASCAL is an ALGOL-like general purpose programming language. It was developed by N. Wirth at the Federal Institute of Technology in Zurich (ETHZ) and was first published in [7]. In 1973 an axiomatic definition of the language followed [4]. Compared to ALGOL 60, PASCAL offers essential extensions in the domain of user defined data structures. In spite of its power of expression PASCAL is characterized by its simplicity and ease of implementation, thereby favourably measuring with other modern programming languages as e.g. PL/I and ALGOL 68.

Parallel with the development and the definition of the language the first PASCAL compiler – for the CDC 6000 computer family – was written at the Computer Science Department of ETHZ during the years 1970 and 1971 [8]. This experience on the one hand led to the definition of a revised language [3] and on the other hand to the decision to write a new compiler from scratch. Thereby, both the language and the implementation matured from the previous project.

The second compiler was started in summer 1972, and completed about two years later. It was developed with the aid of the already existing PASCAL compiler (from now on called the old compiler) using the commonly known bootstrapping technique. This subdivided the whole task into the following four phases:

1) Programming the new compiler in unrevised PASCAL.
2) Compilation of the result of phase 1) with the old compiler.
3) Translation "by hand" of the result of phase 1) into revised PASCAL.
4) Compilation of the result of phase 3) by means of the compiler obtained in phase 2).

The therewith completed separation from the old compiler can be figured with T-diagrams:

The compilers resulting from phase 1) to 4) are herein marked with the respective numbers.

Concerning this bootstrap process, it is worthwhile to note the following:
- The old and the new compiler differ greatly one from the other. This is not so much because of the language revisions but is rather due to the fact that the new compiler generates better code. To reach this goal a total reorganization of the compiler in its code generation parts was necessary and led to a far more complex program structure.
- The compiler resulting from phase 2) obviously compiled revised PASCAL. Nevertheless, it was in two respects not satisfying the demands. First, because it had still been written in unrevised PASCAL and hence could not profit from the language revisions. Second, it had been compiled by the old compiler, which did not translate into sufficiently efficient code. Therefore the compile speed of compiler 2) was rather moderate and its storage requirements were quite high.
- Thanks to the relatively few language changes the hand translation in the third phase of the bootstrap proved to be a more or less negligeable task.
- With the realization of phase 4), the influence of the old compiler on the new one finally vanished.

A compiler bootstrap as performed and described above usually runs into three difficulties:
1) The implementation languages (in this case revised and unrevised PASCAL) have to be powerful enough to express programs of the complexity of a compiler with elegance and ease.
2) The resulting compiler has to produce sufficiently good code. Otherwise it will itself become a victim of its own shortcomings. To be competitive, it must neither be slow nor spacious.
3) Unrecognized errors in the compiler will often become perceptible after a bootstrap in an ugly way. If for example texts of the form f are compiled incorrectly and the compiler itself contains such texts, then, after a bootstrap, the compiler will fail to translate not only these texts correctly but also those texts which are translated by the compiler making use of texts of the form f. Such errors are sometimes very hard to find. Once detected, it can, however, even be more cumbersome to fix them. This is typically the case when, due to the compiler error in question, the intended correction is not correctly compiled.

On the other hand, writing a compiler in a high level programming language has many advantages which nowadays are commonly accepted: saving in programming time, low error rate, easy error detection especially in case of source oriented error messages, and self-documentation - not to speak of (hopefully) improved reliability, modularity and maintainability.

Furthermore, when a compiler is written in the language it
compiles, an additional advantage is that improvements in code
generation often happen to pay back after a bootstrap. In spite
of a larger source text, the compiler is quicker and uses less
memory than before. This feed-back is of course most welcome to
every implementor.

The new PASCAL compiler compiles in one pass. The
characteristics of its syntax analysis are: top-down, one symbol
look-ahead, no backtrack. It is implemented by means of
recursive descent. By using the method of Structured
Programming, the compiler was developed in six explicit steps
that are naturally derived from the process of compilation.

    step 1: Syntax analysis for syntactically correct programs.
    step 2: Recovery from syntactical errors.
    step 3: Analysis of declarations.
    step 4: Treatment of context-sensitive errors.
    step 5: Address assignment.
    step 6: Code generation.

Every step has to be understood as an enrichment of its
predecessor.

It is noteworthy that steps 1 to 4 do not depend on the machine
type for which the compiler is to generate code. Advantage was
taken of this fact during the development of the new PASCAL
compiler for the CDC 6000 computer family. Proceeding from step
4, a compiler for a hypothetical stack computer was also built
[1][5]. An exhaustive description of both compiler projects can
be found in [2].

This report gives a survey of the code generation part of the
CDC 6000 PASCAL compiler. Instead of producing assembler
programs as other compilers do, this compiler generates machine
code directly, therewith providing an efficient compilation
process. To guarantee full flexibility at load time loader
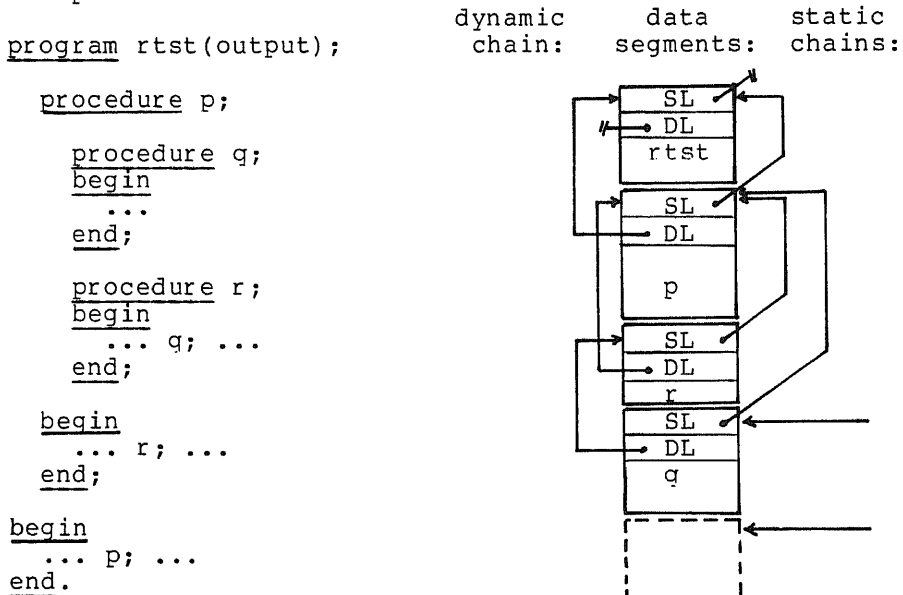compatible relocatable code is generated.

2. The Representation of Data at Runtime

According to a commonly known organization code and data are
separated from each other at runtime. The local data of every
procedure (or function) are united in a data segment and are
therein addressed by an offset relative to the segment origin
(the so called base address). For runtime a stack containing the
data segments of all activated procedures is provided. As the
base addresses of the data segments vary during runtime,
variable addressing is nontrivial. However, this way to organize
data guarantees maximum storage economy: Every data segment

exists only during the execution of the procedure to which it belongs. It is created at procedure entry and discarded at exit. Both of these stack operations can be implemented at relatively low expense.

To allow stacking and unstacking of data segments, a link is needed. It is called dynamic link (DL) and chains every data segment to its immediate predecessor in the stack. Variable addressing is done through a second link. It is called static link (SL) and chains only those data segments which - according to the scope rules of PASCAL - are currently accessible. SL as well as DL are incorporated in the head of every data segment (as first and second word respectively).

Example:

program rtst(output);

   procedure p;

     procedure q;
     begin
       ...
     end;

     procedure r;
     begin
       ... q; ...
     end;

   begin
    ... r; ...
   end;

begin
  ... p; ...
end.

dynamic chain: data segments: static chains:

program   and   corresponding stack (growing downwards) of data segments belonging to the calling sequence rtst --> p --> r --> q

BASE is the base address of the most recently created data segment. It is the head of the two chains and is hence used both for addressing variables and for the two stack manipulations. Stacking also requires another pointer (NEXT) defining the base address of the segment to be stacked.

As shown in the following figure a data segment is, in general, divided into six logically distinguishable areas.

```
offset with respect
to segment origin:
        Ø  ┌──────────────────────┐
        1  │       segment        │
           │        head          │
        2  ├──────────────────────┤
        3  │   function result    │
           ├──────────────────────┤
        .  │     parameter        │
           │    descriptors       │
      n+2  │                      │
      n+3  ├──────────────────────┤
        .  │                      │
           │   parameter copies   │
        .  │                      │
           ├──────────────────────┤
        .  │   local variables    │
           ├──────────────────────┤
           │  anonymous values    │
           └──────────────────────┘
```

<u>segment head</u>: It contains the two links SL and DL described above.

<u>function result</u>: This word is reserved for the eventual function result. In data segments belonging to procedures it is not used.

<u>parameter descriptors</u>: For each of the n parameters a descriptor word is reserved; thereby the descriptor of the i-th parameter gets an offset of i+2 . The information stored therein depends on the kind of the parameter. For those value parameters needing at most one word of memory, the descriptor contains the actual value. For variable parameters and for value parameters needing more than one word of memory, the descriptor is used to hold the (first word) address of the actual parameter. Finally, in case of a parameter procedure the descriptor consists of its entry point and static link.

<u>parameter copies</u>: Local copies of those value parameters which need more than one word of memory are stored in this area.

<u>local variables</u>: This area is reserved for the values of the local variables.

<u>anonymous values</u>: In this last section of a data segment anonymous values are temporarily stored (e.g. upper bounds of for-loops, which - according to the definition of PASCAL - are computed only once).

Any of the last four areas may be absent in a particular data segment.

The following example shows a function declaration and the data segment derived therefrom:

offset:

```
const n = 3;
type inx = 1..n;
     arr = array [inx] of real;

function scalprod(a,b: arr;k: inx): real;
   var s: real; i: inx;
begin s := 0;
   for i := 1 to k do s := s + a[i]*b[i];
   scalprod := s
end
```
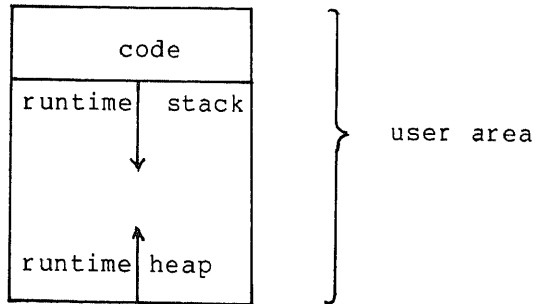
	offset
SL	0
DL	1
scalproc	2
address of a	3
address of b	4
value of k	5
value of b	6
	7
	8
value of a	9
	10
	11
value of s	12
value of i	13
loop bound value(k)	14

After completion of the loading process a contiguous piece of yet unused store remains at the upper end of the user area. During execution this piece of storage is used for the runtime stack as well as for the runtime heap. The former grows from the lower end while the latter uses the upper part and grows in the opposite direction.

```
+-------------------+
|       code        |
+-------------------+   }
| runtime   stack   |   |
|         |         |   |
|         v         |   }   user area
|                   |   |
|         ^         |   |
| runtime | heap    |   }
+-------------------+
```

3. The Compilation of Expressions

3.1. The Data Structure Describing Partially Compiled Expressions

The expression or subexpression currently in compilation is described by a global variable called GATTR (global attribute record) of type ATTR (to be defined later).

Whenever a dyadic operation has to be compiled the attributes of the first operand are stored in a local variable LATTR after its

compilation. LATTR is, of course, local to that one procedure which translates the operation (e.g. for a multiplication, local to the compiler procedure TERM). After the compilation of the second operand, the attributes of both operands are available in LATTR and GATTR respectively - thereby allowing the operation to be compiled.. Besides defining the attributes of the result in GATTR, this action generally involves code generation.

Example:

```
procedure term;
     var lattr: attr; lsymbol: pascalsymbol;
begin factor;
     while symbol in mulops do
          begin lattr := gattr; lsymbol := symbol;
               readnextpascalsymbol; factor;
               case lsymbol of
                 mulop: generate multiplication code      1)
                        and define attributes of result
                 divop: ...
                         .
                         .
                         .
             end (*case*)
        end (*while*)
end (*term*)
```

1) A rectangle ⬛x occurring in a piece of program usually stands for a statement performing the computation which is verbally defined by x. However, it may - as in the next example - also stand for the declaration of data described by x.

In the attribute record ATTR the following four classes of expressions are distinguished: constants (CST), variables (VARBL), conditions (COND), and other expressions (EXPR). With a minimum of discriminations this classification allows the generation of efficient code.

Hence, the type ATTR has the following form:

```
     attr = record typtr: pointertotypeentry;
                  case kind: attrkind of
                    cst:   attributes of the constant   ;
                    varbl: attributes of the variable    ;
                    cond:  attributes of the condition   ;
                    expr:  attributes of the expression  ;
             end
whereby attrkind = (cst,varbl,cond,expr).
```

The following remarks are destined to give some insight into this data structure and its usage.

TYPTR: holds the pointer to the type entry. It is used not only

for type checking but also for code generation. On the one hand it allows the access to size information (i.e. the number of words and bits associated with the type). On the other hand it makes available index and subrange bounds which are needed for address calculations and assignment checks.

KIND = CST: The only additional attribute of a constant is its internal value (CVAL).
In connection with some operations, constant operands give rise to a special treatment. This applies to
a) Indexing:
   The address calculation is done at compile time.
b) Integer multiplication:
   If one of the operands has the form $c = 2^{**}n$ (power of two) and $n>2$, a shift instruction is generated. However, for multiplications by two ($n = 1$) an add instruction is compiled:

> if $n \neq 0$ then
>   if $n = 1$ then   1)
>   begin $\boxed{\text{IXj Xi+Xi}}$; i := j end
>   else $\boxed{\text{LXi n}}$
>   (* result register is Xi *)

1) For the CDC 6000 instruction set cf. [9].

The reason is that the shift destroys the other operand (in Xi), while the add instruction leaves it untouched, thereby enriching the register contents.

For constants of the form $c = 2^{**}n*(2^{**}m \pm 1)$ with $m \geq 1$, i.e. for sums and differences of powers of two, the code generation is defined by

> if $n \neq 0$ then $\boxed{\text{LXi n}}$;
> $\boxed{\begin{array}{ll} \text{BXj} & \text{Xi} \\ \text{LXj} & \text{m} \\ \text{IXj} & \text{Xj} \pm \text{Xi} \end{array}}$
>   (* result register is Xj *)

These trivial optimizations reduce the execution time considerably. While loading c and doing a multiply instruction takes 6.3 microseconds the optimized code is executed in 0.6 to 2.3 microseconds. It is noteworthy that only three of the integers up to 20 (namely 11, 13, and 19) are not of one of the distinguished forms.

This code improvement is most important for the hidden multiplications that have to be compiled in order to address components of structured variables. Experience shows that such code is frequently executed.
c) The operations div and mod:
   When the second operand is a power of two, both operations

are compiled with shifts.
d) Checks verifying the legality of an operation:
   Example: Indexing is an illegal operation if the actual index
   is out of the declared index range. Thus, given the
   declaration
        a: array[1..10] of integer,
   a[0] is illegal and is diagnosed at compile time. Similar
   checks hold for such operations as assignments to variables
   of subrange type and integer division (dividend $\neq$ 0).

KIND = VARBL: The three kinds of access naturally given by the
language split this class into three subclasses. The
discriminating field WORDACC is of type
     accesskind = (drct,indrct,inxd).
The value of WORDACC specifies the access to the word containing
the variable as follows:
   WORDACC = DRCT: (direct access) The word address is
   represented by the pair <declaration level, relative address>
   (<VLEVEL,CWDISPL>). VLEVEL determines which data segment has
   to be accessed and CWDISPL holds the offset relative to the
   segment origin.
   Examples: entire actual variables, entire value parameters,
   and record components.
   WORDACC = INDRCT: (indirect access) The word address is
   obtained by adding an offset (CWDISPL) to the value of an X
   register (the one with index VWDISPL).
   Examples: variable parameters, file elements, variables
   referenced through pointers.
   WORDACC = INXD: (indexed access) The word address is composed
   from an array address given by the pair <VLEVEL,CWDISPL>, and
   the value of the X register with index VWDISPL. The sum of
   these two address components yields the word address of the
   variable.
   Example: actual array elements.
   In the case of packed variables another field (CBDISPL) is
   needed to hold the constant part of the bit address. With
   indexed access to packed variables, CBDISPL has to be added
   to the value of the X register with index VBDISPL to get the
   bit address.
   Notice that indirect access to packed variables does not
   exist in this implementation. The reason for this is
   threefold.
   First: as an implementation restriction, no component of a
   packed variable may be substituted for a variable parameter.
   Without this restriction either packed variables or variable
   parameters could not be implemented in a sensible way.
   Second: The only files for the components of which less than
   one word of memory is allocated are the text files (of type
   packed file of char). However, for these files the PASCAL
   runtime support provides and manages a character buffer
   wherein the actual file element appears in unpacked form.
   Third: The memory allocation for variables created at runtime
   is such that these variables always occupy an integral

multiple of words. Thus, pointers can be implemented as <u>word</u> addresses.

<u>KIND = COND</u>: In accordance with the user defined scalars, the constants FALSE and TRUE of the standard scalar
      Boolean = (false,true)
are internally represented by the values 0 and 1 respectively. This representation is the only one that allows straight-forward code when Boolean expressions appear in case-statements, for-statements, index expressions, etc.
Without the case COND, every relation (e.g. i>j) would have to be compiled up to the Boolean result – regardless whether this was really necessary.

Example: For "<u>if</u> i > j <u>then</u> ..." the following code would have to be generated

```
SA1    i
SA2    j
IX0    X2-X1
MX3    1
BX0    X3*X0
LX0    1
ZR     X0,...
```

This is obviously far from optimal and can easily be done better. Instead of hastily compiling the relational operation up to the Boolean result, only the subtraction instruction is generated. The index of the condition register (0 in the above example) is kept in CDR and the field CONDCD: (ZR,NZ,PL,NG) is set to the condition code (here PL). Later, when it becomes obvious whether the Boolean value is needed or not, the appropriate code is generated. In the above example a "PL X0,..." would replace the last four instructions. (Note that in this implementation −0´s are eliminated by adding a +0 whenever a −0 could possibly result. Consequently, jumps depending on the sign bit only are safe.)

The compilation of the Boolean operator <u>not</u> turns out to be trivial for KIND = COND. No code need be generated; CONDCD has merely to be updated.

```
case condcd of
   zr: condcd := nz;
   nz: condcd := zr;
   pl: condcd := ng;
   ng: condcd := pl
end
```

It is worthwhile noting that not even the Boolean result of a relation is needed when the relations appear as operands in one of the operations <u>and</u> or <u>or</u>. An examination of the 12 possible

combinations of CONDCD shows that in every case better code can be produced for both operations.

Example: <u>if</u> (i > j) <u>and</u> (k ≠ 3) <u>then</u> ... is compiled into

```
SA1    i        ⎫
SA2    j        ⎬  i > j
IXØ    X2-X1    ⎭
SA3    k        ⎫
SX4    3        ⎬  k ≠ 3
IX5    X3-X4    ⎭
BX6    X6-X6    ⎫
IX6    X6-X5    ⎬  and
BX5    X5-X6    ⎭
BX6    XØ*X5
PL     X6,...
```

<u>KIND = EXPR</u>: All other expressions fall into this class. The only attribute is the index EXPREG of the X register, which at runtime contains the value of the expression.

Now, the complete definition of ATTR can be given:

```
attr = record typtr: pointertotypeentry;
            case kind: attrkind of
                cst: (cval: value);
                varbl: (wordacc: accesskind;
                        vlevel: levrange; cwdispl: integer;
                        vwdispl: regnr;
                        case pckd: Boolean of
                        true: (cbdispl: integer;
                                    case bitreg: Boolean of
                                    true: (vbdispl: regnr);
                                    false: ());
                        false: ());
                cond: (cdr: regnr; condcd: (zr,nz,pl,ng));
                expr: (expreg: regnr)
            end
wherein regnr = Ø..7.
```

A couple of examples of expressions are collected in table 1. The first column shows the expression which is to be compiled by a call of the compiler procedure EXPRESSION. The second column contains the attribute record thereby created and the third column shows the code generated during the call. The underlying declarations are

```
var i,j: integer; p: ↑integer;
    a: array [1..2] of char;
    r: packed record f1: Boolean;
                     f2: packed array [1..4] of char
            end
```

The declaration level is assumed to be 1, the relative addresses range from 3 (for i) to 8 (for r).

resulting compiler internal description of the expression (to be found in GATTR after return from EXPRESSION)

expression to be compiled	typ	kind	cval/ wordacc/ cdr/ expreg	vlevel/ condcd	cw-displ	vw-displ	pckd	cb-displ	bit-reg	vb-displ	code generated during the call of EXPRESSION
true	boolptr	cst	1								—
i	intptr	varbl	drct	1	3	—	false				SA1 i / LX1 1
a[2]	charptr	varbl	drct	1	7	—	false			—	SA1 i
a[i]	charptr	varbl	inxd	1	5	e.g.2	false				SA2 i (i)
p↑	intptr	varbl	indrct	—	0	e.g.1	false				SA1 p (p)
r.f2[i]	charptr	varbl	drct	1	8	—	true	-5	true	eg.6	IX6 X6+X1 (6*i) / LX6 1 / BX6 X1 / LX1 1 / SA1 i
i + 10	intptr	expr	e.g.3								IX3 X2+X0 (i+10) / SX0 10 / SA1 i
i > j	boolptr	cond	e.g.0	p1							IX0 X2-X1 (j-i) / SA2 j / SA1 i

Table 1: Examples of compiled expressions

Remarks:
- The register indexes are incidental. In a concrete compilation their choice depends on the actual register contents (cf. 3.2.-3.4.).
- In the example "r.f2[i]" the value of WORDACC (DRCT) must not surprise. The word address of this variable is independant of the value of i.

3.2. The Data Structures Describing the Register Contents

The fact that the target machine offers 24 hardware registers represents a challenge to the compiler writer. He can expect that book-keeping of the register contents will prevent him from generating many superfluous instructions. Of course he must carefully weigh for what classes of expressions book-keeping pays off in so far as it is sufficiently manageable and code is considerably improved.

In this one-man compiler project the simplifications in code generation are restricted to load operations:
- The generation of load instructions is unnecessary when an inspection of the description of the register contents shows that the value of the expression in question is already in a register.
- A load operation is reduced to a single 15 bit instruction when an inspection of the register contents shows that a register holds the address in question or one of its neighbouring addresses.

Clearly, the compiler also uses this information to aim at rich register contents. Whenever a register is needed, a run through the description of all the registers is performed. Thereby the compiler decides which register can be released with supposedly least harm to further code generation (for details cf. 3.3.).

The following paragraphs intend to give insight into the compiler data structures describing the contents of the X , A, and B registers. With respect to the compiler complexity it was decided that only those classes of expressions which can be described with at most two unstructured values are worth remembering.

3.2.1. The Contents of the X Registers

The simplest category of expressions are the constants. They are completely defined by their internal value. However, for the code generation, it is suitable to subdivide the set of all constants into two classes: the "short" constants (SHRTCST), and the "long" constants (LONGCST). The former can be loaded with fast SX instructions while the latter must be loaded with slow SA instructions.

No other register contents are described with the same ease. For another class of expressions, the pair <declaration level, relative address> met above is the appropriate description. These are called the simple variables (SIMPVAR). It is noteworthy that not only user defined variables but also the parameter descriptors (thus also addresses) and anonymous values fall under this notion.

Finally, a fourth class of register contents is worth remembering: the so called indirect variables (INDVAR). They are

described by a pair <X register number, relative address>. The
first component references an X register with contents SIMPVAR
and the second represents the offset. The sum of the two address
components is the word address of the indirect variable. This
class comprises variable parameters (offset 0), file elements,
and variables that are referenced through pointers.
An X register, occupied by an intermediate result not belonging
to one of the classes distinguished, is characterized by OTHER.
A free register is marked AVAIL.

Every register not being AVAIL has a counter (REFNR), which
indicates how many times the register is currently being
referenced. The references either originate from the description
of another X register (cf. INDVAR above), or from a variable of
type ATTR. As long as a register is referenced it must obviously
not be overwritten.

When no references exist, it is interesting to know for how long
the register has not been referenced. An easy measure for this
is the difference between the current value of the instruction
counter and the one at the moment of the expiration of the last
reference (LASTREF). The longer a register has not been
referenced, the less interesting it is to the compiler to
protect its contents from being overwritten.

Both simple and indirect variables can be packed. In such cases
it will often happen that an X register is rotated by a number
of bits because the last action has been the unpacking of a
component of the (structured) variable. A shift count (SHFTCNT)
is therefore added to the description of the two classes SIMPVAR
and INDVAR. This shift count tells by how many bits the register
content is actually left-rotated.

After these introductory explanations the data structure for the
description of the contents of the X registers can now be given.

```
type xrgclass = (avail,shrtcst,longcst,simpvar,indvar,other);
     xrgclassl = shrtcst..indvar; xrgclass2 = simpvar..indvar;
     xrgcont = packed record
                     case xcont: xrgclass of
                       avail: ( );
                       shrtcst,longcst,
                       simpvar,indvar,
                       other: (refnr: shortint; lastref: icrange;
                                  case xrgclassl of
                                    shrtcst: (cstval: shortint);
                                    longcst: (cstptr: ↑csttable);
                                    simpvar,
                                    indvar: (shftcnt: 0..59;
                                                case xrgclass2 of
                                                  simpvar:
                                                    (xlev: levrange;
                                                     xaddr: addrrange);
                                                  indvar:
                                                    (xreg: regnr;
                                                     xdispl: addrrange);
                     end;
     xrgconts = array [regnr] of xrgcont;
 var xrgs: xrgconts
```

XRGS always describes the register contents corresponding to the
actual state of the code generation.
Example: The compiler internal descriptions of expressions given
in table 1 reference (among others) the registers X2 and X6. The
descriptions of the contents of the two registers are given
below.

register	content	description in XRGS				
		xcont	refnr	shftcnt	xlev	xaddr
X2	i	simpvar	1	0	1	3
X6	6*i	other	1			

Note that with this kind of description the compiler remembers
not only unstructured variables (i.e. those of scalar, subrange,
or pointer type) but also structured ones (i.e. those of set,
array, or record type). For example as soon as a field of the
record r (cf. the declarations underlying table 1) has been
loaded, the whole record is in a register. Any succeeding access
to one of its fields can therefore be performed without memory
reference.

3.2.2. The Contents of the A Registers

The addresses which are remembered are exactly those of the
variables which are remembered. Hence, three kinds of addresses
are distinguishable, those of simple variables (SIMPADDR),
indirect variables (INDADDR), and others (UNSPECADDR).

```
type argclass = (simpaddr,indaddr,unspecaddr);
     argcont = packed record
                   case acont: argclass of
                     simpaddr: (alev: levrange;adispl: addrrange);
                     indaddr: (areg: regnr; adispl: addrrange);
                     unspecaddr: ( )
                   end;
     argconts = array [regnr] of argcont;
 var args: argconts
```

ARGS also describes the register contents as they correspond to the actual state of the code generation.

3.2.3. The Contents of the B Registers

The uses of the B registers are manifold. Their usage as base registers for the data segments is of course the most important one. In this connection the question arises as to which of all possible methods of access to data segments should be implemented. In principle, access is possible through the static link. However, for any nonlocal variable this kind of access is indirect and therefore inefficient. A better solution consists in having all active base addresses directly accessible. This was done in an earlier implementation of PASCAL [8] in an extremely effective way by using the B registers as a hardware display. The overhead of this method is modest: at procedure entry, a B register has to be saved before it is loaded with a new value; at procedure exit, the old value has to be restored. The disadvantage is rather that the limited number of available B registers imposes a severe restriction on the degree of procedure nesting. In a high level programming language implementation this is absolutely undesirable.

In this implementation access to the data segments is therefore organized differently. Register B2 is permanently used to hold the base address of the main program variables. Besides this, only the base address of the data segment belonging to the procedure actually in execution resides in a B register, namely in B5. To insure fast stack and unstack operations of the data segments B6 is reserved for NEXT(cf. 2.). Again, for reasons of efficiency B4 always points to the "first" word in the heap. This eases the test on an eventual overlap of stack and heap as well as the implementation of the standard procedure NEW.
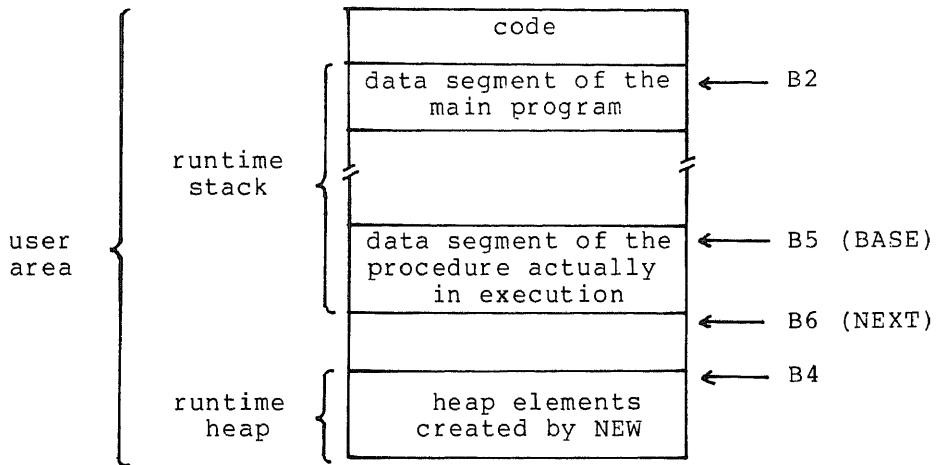
The remaining three B registers - except for B0 - are used as follows:
B1 always contains the constant 1. This allows a considerable condensation of the generated code in so far as in many places a 15 bit instruction can be generated instead of a 30 bit instruction. In such cases not only the length of the code is reduced to half but also the probability that fill instructions have to be generated is diminished.
The advantages of letting B1 = 1 are manifold:
- It happens relatively often that variables which are declared together - and therefore get adjacent addresses - are used in the same context. Obviously, this leads to the usage of neighbouring addresses in these program parts.
- Count variables are usually incremented or decremented by 1. In for-loops this happens even implicitly.
- Except for 0, 1 is the most frequent initial value.
- Comparisons and assignments of storage areas are done word by word using an address increment of 1.
- The constant 2 (=B1+B1) is another frequently used constant.

Finally, B3 and B7 are used for miscellaneous purposes but particularly for variable addressing. It was mentioned above that the main program variables are always directly accessible (through B2) just the same as are the local data of the procedure in execution (through B5). Now, whenever access to a data segment of intermediate level first occurs, it is achieved by following the static link. However, if one of the registers B3 or B7 is currently available, this register is immediately used to point to the origin of the data segment in question. Subsequent accesses to this data segment are then direct, i.e. without the detour along the static link. There is a good chance that indirect accesses can thereby be considerably reduced, if not nearly eliminated.

To implement this method of segment addressing the following variables were introduced

```
          levels: set of levrange;
          brg: array [levrange] of regnr
```

LEVELS is the set of those declaration levels of which the data
segments are currently accessible through B registers. Thus, it
always contains the set [1,LEVEL]. BRG maps these levels into
the corresponding B register indexes and is initialized by

```
          brg[1] := 2; brg[level] := 5.
```

Access to a data segment of level 1 is now as follows:

```
     if 1 in levels then direct access through Bbrg[1]
                    else for k := 1 + 1 to level do
                             if k in levels then
                                 indirect access through Bbrg[k]
                                 down the static link. If a B re-
                                 gister (Bn) is free, let it hold
                                 the base address of the accessed
                                 data segment and update LEVELS
                                 and BRG:
                                 levels := levels + [1];
                                 brg[1] := n
```

This kind of organization of variable addressing is quite
modular. For example, it would be very easy to return to the
hardware display method (levels := [1..level]; for 1 := 1 to
level do brg[1] := 1) implemented in the earlier compiler
version. As a consequence, L in LEVELS would be true for all L.
Hence the else-part of the above statement would never have to
be executed.

To distinguish between the several uses of the B registers, the
type
        brgclass = (free,basaddr,specpurp)
was introduced in the compiler. FREE characterizes a B register
which is currently not used. B registers containing nonpermanent
base addresses are described by BASADDR, and SPECPURP indicates
a special mode of use that forbids overwriting.

The complete declaration of the variable BRGS, describing the
actual state of the B registers, follows.

```
     type brgcont = packed record
                         case bcont: brgclass of
                             free: ();
                             basaddr: (blev: levrange);
                             specpurp: ()
                     end;
          brgconts = array [regnr] of brgcont;
     var brgs: brgconts
```

For the registers B0, B1, B2, B4, B5 and B6 the field BCONT
always has the value SPECPURP while for B3 and B7 its value
varies according to the actual mode of use.

3.3. The Register Allocation Procedures

The procedures involved in code generation must be given the possibility to require registers. As to the A and X registers, which are coupled (cf.[9]), one procedure (NEEDX) is sufficient. Its input parameters define the interval in which the output parameter, i.e. the index of the disposed register, has to lie. Four kinds of calls have to be distinguished:

    needx(0,7,i): A call asking for an arbitrary X register.
    needx(1,5,i): A call asking for an A/X register pair to
                  compile a load operation.
    needx(6,7,i): A call asking for an A/X register pair to
                  compile a store operation.
    needx(i,i,i): A call asking for the register Xi (e.g. X6
                  to pass a function result to the calling
                  procedure).

Similarly, B registers are requested through a call of NEEDB. This procedure . however, has only one parameter, the index to be returned.

Both NEEDX and NEEDB choose the register to be disposed depending on the actual register contents as described by the variables XRGS and BRGS. These variables are updated immediately after the choice in order to mirror the new register contents correctly.

In the case of NEEDX the selection algorithm depends only on the contents of the X registers which are considered to be far more valuable than the corresponding A register contents. If among the registers in question there is one with XCONT = AVAIL, its index is returned. Otherwise that one register (with REFNR = 0) for which the value of BONUS[XCONT]-LASTREF is a maximum is disposed.

For this register the field XCONT is set to OTHER and REFNR is set to 1.

The array BONUS is defined as follows:

    bonus[shrtcst] := 20; bonus[longcst] := 10;
    bonus[simpvar] := 4; bonus[indvar] := 3;

This initialization assigns to the short constants the least importance. The reason for this is that their occurrences are rarely clustered, and they can be loaded with fast SX instructions. The long constants are weighed more heavily, but still subordinated to the variables. This is because the probability of frequent use of the same long constant is expected to be much lower than frequent use of the same variable.

Among registers containing simple or indirect variables the choice depends heavily on the "time" of the last reference

(LASTREF).

A register which has been requested and is not used any more has to be released. In the case of the X registers this is done by calling the procedure DECREFX. Its parameter is the index of the register to be returned. DECREFX decrements the number of references REFNR by one. Once it has reached zero, LASTREF is set if XCONT ≠ OTHER, otherwise XCONT gets the value AVAIL.

Hence, in the compiler, typical code generation sections using an X register have the form

        needx(0,7,i); code generation involving Xi ; decrefx(i)

Returning a B register is even more trivially accomplished by assigning the value FREE to BCONT. Code generation sections making use of a B register therefore show the following pattern:

  needb(k); code generation involving Bk ; brgs[k].bcont := free

3.4. Loading an Expression

Generating code to load the value of an expression into an X register is a fundamental operation and has to be done in many places in a compiler. This fact evidently calls for the introduction of a procedure (LOAD) to perform the necessary code generation and the associated updating of the register contents. Therein, the description of the expression to be loaded appears as an input parameter, and the index of the X register containing the value at completion of the load operation is the output parameter. The global register content descriptions ARGS, XRGS, and BRGS are transient. The structure of the variable FATTR (describing the expression to be loaded) determines the layout of LOAD:

```
procedure load(fattr: attr; var fi: regnr);
  var i: regnr;
begin
  with fattr do
    if typtr ≠ nil then
      case kind of
        cst:
          either find a register Xi containing the value described
          by CVAL or generate code to load that value into Xi; up-
          date the register content description accordingly
        varbl:
          begin
            case wordacc of
              drct:
                either find a register Xi containing the value of
                the memory word described by <VLEVEL,CWDISPL> or
                generate code to load that word into Xi; update
                the register content description accordingly
              indrct:
```

```
                        either find a register Xi containing the value of
                        the memory word described by <CWDISPL,VWDISPL> or
                        generate code to load that word into Xi; update
                        the register content description accordingly
                  inxd:
                        generate code to load the memory word described
                        by <VLEVEL,CWDISPL,VWDISPL> into Xi; update the
                        register content description accordingly
                  end (*case*);
                  if pckd then
                        begin
                        by consulting the type of the variable deter-
                        mine:
                        - the number of bits occupied by the variable
                        - whether a sign bit has to be extended
                        - whether the value has to be left or right
                          justified (the former in case of record
                          and array variables)
                        if bitreg then
                              generate code rotating the value with the bit
                              address <CBDISPL,VBDISPL> into the appropriate
                              position for unpacking
                        else
                              generate code rotating the value with the bit
                              address CBDISPL into the appropriate position
                              for unpacking
                        generate code unpacking the value of the
                        variable into another X register and store
                        its index in i
                        end (*pckd*)
                  end (*varbl*);
            cond:
                  depending on the value of the condition code CONDCD
                  generate code to compute the Boolean value from the
                  value of the condition register Xcdr
            expr:
                  i := expreg
            end (*case*)
      else needx(0,7,i);
  fi := i
end (*load*)
```

As a further refinement would be too detailed, the description
of the procedure LOAD is concluded by summarizing the code which
- depending on FATTR - is generated. It is to be noted that
since the different cases are not mutually exclusive, their
order is relevant.

KIND = CST:

```
1. abs(cval) > 2**17          SAi @cval
2. cval = 0                   BXi Xi-Xi                          1)
3. cval = 1                   SXi B1
4. cval = 2                   SXi B1+B1
5. else                       SXi cval
```

KIND = VARBL, WORDACC = DRCT:

1. address already in Aj   SAi Aj
2. neighbouring address
 already in Aj   SAi Aj+B1
3. vlevel <u>in</u> levels   SAi Bbrg[vlevel]+cwdispl   2)
4. else   ┘ code to load the base address
           of the data segment with 3)
           level VLEVEL into Xi
           SAi Xi+cwdispl   2)

KIND = VARBL, WORDACC = INDRCT:

1. address already in Aj   SAi Aj
2. neighbouring address
 already in Aj   SAi Aj+ B1
3. cwdispl = 0   SAi Xvwdispl
4. cwdispl = 1   SAi Xvwdispl+B1
5. else   SAi Xvwdispl+cwdispl

KIND = VARBL, WORDACC = INXD:

1. vlevel <u>in</u> levels
 1.1 three quarters of the
   actual code word   SXi Xvwdispl+Bbrg[vlevel] 4)
   are filled   SAi Xi+cwdispl   5)
 1.2 else   SXi Xvwdispl+cwdispl   5)
          SAi Xi+Bbrg[vlevel]
2. else   ┘ code to load the base address
           of the data segment with 3)
           level VLEVEL into Xi
           IXi Xi+Xvwdispl   6)
           SAi Xi+cwdispl   5)

KIND = COND:

1. condcd <u>in</u> [pl,ng]   MXK 1
           BXi(-)Xcdr*Xk   7)
           LXi 1
2. condcd <u>in</u> [zr,nz]   BXi Xi-Xi
           IXi Xi-Xcdr
           BXi(-)Xi-Xcdr   8)
           MXk 1
           BXi Xi*Xk
           LXi 1

Remarks:
1) The execution time for BXi Xi-Xi is 0.5 microseconds compared
 to 0.6 microseconds for a SXi B0 (CDC 6400).
2) CWDISPL <u>in</u> [0,1] is impossible because the first relative
 address assigned in every data segment is 2. Therefore this
 instruction is always 30 bits.
3) The procedure LOADBASE performs this task. Therein a search
 through BRGS is done to find the index k of the B register

containing the base address of the data segment with level 1
for which 1 - VLEVEL is a minimum. Then the following code is
generated:

```
        SAi  Bk
        SAi  Xi  ⎫
          .        ⎬   1 - vlevel - 1   instructions
          .        ⎭
        SAi  Xi
```

which is succeeded by a SBn Xi when there is an n with
BRGS[n].BCONT = FREE.
4) This avoids the generation of a fill instruction (NO).
5) CWDISPL in [0,1] is possible but improbable. A special
   treatment to prevent a NO instruction would hardly pay off.
6) Alternative code would be

```
        SBk  Xi+cwdispl
        SAi  Xvwdispl+Bk.
```

7) The minus is valid for CONDCD = NG.
8) The minus is valid for CONDCD = NZ.

In case of a packed variable further code is necessary to unpack
the relevant bits from Xi into another X register. To achieve
this, a variable

```
        lmode: (usradj, sradj, usladj)
```

is introduced to distinguish between three possibilities.
LMODE = USLADJ: The variable is of record or array type. Hence,
   the value is structured, therefore unsigned, and has to be
   left adjusted. It is not necessary to mask out the bits not
   belonging to the value.
LMODE = SRADJ: The variable is of subrange type, the lowbound of
   which is less then zero. The value is signed and has to be
   right adjusted.
LMODE = USRADJ: The variable has another type. The value in this
   case is unsigned and has to be right adjusted.

The corresponding code generation is defined by the next program
piece. Request and release of the registers as well as the
updating of the register content description are intentionally
left out for easier reading.

```
with fattr do
  begin
    with typtr↑ do
      begin
        if form = subrange then
          if min.ival < 0 then lmode := sradj
                          else lmode := usradj
        else if form in [arrays,records] then lmode := usladj
                                          else lmode := usradj;
        bitsz := size.bits
      end;
    with xrgs[i] do
      if xcont in [simpvar,indvar] then shift := shftcnt
                                   else shift := 0;
    mask := wordsize - bitsz; cshft := cbdispl - shift;
    if lmode = usradj then cshft := cshft + bitsz
```

```
      end ;
  if bitreg then
      begin
          if shift ≠ 0 then
              begin LXi wordsize-shift ; cshft := cshft + shift end;   1)
          if cshft in [0,1] then SBk Xvbdispl+Bcshft
                             else SBk Xvbdispl+cshft ;
          LXj Bk+Xi
      end
  else
      begin
          with xrgs[i] do
              if (xcont in [simpvar,indvar]) and (lmode = sradj)
                  then BXj Xi                                          2)
                  else j := i;
          if cshft ≠ 0 then LXj cshft
      end;
  case lmode of
      usradj: begin MXk mask ; BXk -Xk*Xj end;
      sradj:  AXj mask ;
      usladj:
  end
```

Remarks:
1) This shift is necessary as otherwise Bk could be set to a negative value. According to the CDC 6000 hardware specification the succeeding shift would then become a (sign extending) arithmetic right shift and would lead to a wrong result.
2) The register contents are copied to avoid destruction in the succeeding shift operation.

Table 1 showed the effect of a call of EXPRESSION for the variable r.f2[i]. The internal description produced and the code generated were therein given as summarized below.

internal description in GATTR	generated code	description of the register contents					
		X registers			A registers		
		X	xcont	refnr	A	acont	alev	aaddr
typtr: charptr	1)							
kind: varbl	SA1 B2+3	1	avail		1	simp-addr	1	3
wordacc: drct	LX1 1	:						
vlevel: 1	BX6 X1							
cwdispl: 8	LX6 1	:						
vwdispl: -	IX6 X6+X1	6	other	1				
pckd: true								
cbdispl: -5	i.e.							
bitreg: true	X6 := 6*i							
vbdispl: 6								

A succeeding LOAD(GATTR,I) would have the following effect:

internal description in GATTR	generated code	description of the register contents								
		X registers					A registers			
		X	xcont	refnr	xlev	xaddr	A	acont	alev	aaddr
	1)	0	avail							
	SA2 B2+8	1	avail				1	simp-addr	1	3
	SB3 X6+B1									
typtr: charptr	LX0 B3,X2	2	simp-var	0	1	8	2	simp-addr	1	8
kind: expr	MX3 54									
expreg: 3	BX3-X3*X0	3	other	1						
	i.e.X3:=	6	avail							
	r.f2[i]									

1) if not already loaded.

3.5. The Compilation of Operations

In chapter 3.1. the compiler procedure TERM, in which the
operations *, /, div, mod, and and are compiled was introduced.
At that time, the associated code generation was not given since
the prerequisites for understanding it were missing. This is no
longer the case, and multiplication can be taken as an example.

Under the assumption that neither of the factors is a constant,
which would give rise to optimized code (cf. 3.1.), the
following program piece compiles the multiplication operation:
     load(lattr,i); load(gattr,j);
     decrefx(i); decrefx(j); needx(0,7,k);
     IXk Xi*Xj ;
     with gattr do
        begin typtr := intptr; kind := expr; expreg := k end
The first factor (described by LATTR) is loaded into an X
register. By means of the variable i its index is passed to
TERM. Analogously the second factor (described by GATTR) is
loaded into Xj. If a factor is recognized as being already
loaded, LOAD does not, of course, generate code.

From the load operation each of the registers Xi and Xj gets a
reference (REFNR = 1). However, both of them are immediately
erased through calls of DECREFX because the contents of Xi and
Xj need not be protected any longer. With the subsequent call of
NEEDX the result register for the multiplication is required.
Due to the previous erasure of the references to Xi and Xj, it
is well possible that the returned register index k is equal to
i or j. The above calling sequence is therefore preferable to
     ... needx(0,7,k); decrefx(i); decrefx(j); ...
which restricts the freedom of choice of the selection algorithm
in NEEDX.

Finally, the multiplication instruction is generated and the
internal description of the result GATTR is defined.

A second example shows the process of code generation for the
expression (i > j) and (i >= 10). First, the left hand factor of
this term is compiled (cf. table 1).

generated code: internal desciption of the result:

```
SA1 B2+3                     typtr: boolptr
SA2 A1+B1                    kind:      cond
IX0 X2-X1                    cdr:          0
                            condcd:       pl
```

Then, the right hand factor:

```
SX3 10                       typtr: boolptr
IX4 X1-X3                    kind:      cond
                            cdr:          4
                            condcd:      ng
```

Finally, the two factors are disjunctively joined in TERM.

```
BX0 -X4*X0                   typtr: boolptr
                            kind:      cond
                            cdr:          0
                            condcd:       pl
```

This terminates the compilation of the conjunction. The resulting register contents are described by XRGS and ARGS as follows:

X	xcont	refnr	lastref	xlev/cval	xaddr	A	acont	alev	aaddr
0	other	1							
1	simpvar	0		1	3	1	simpaddr	1	3
2	simpvar	0		1	4	2	simpaddr	1	4
3	shrtcst	0		10					
4	avail								

Of course, TERM could load the two factors with a call to LOAD. In that case code would be generated to determine the associated Boolean values (0 for FALSE, 1 for TRUE). However, this code would be very inefficient. Instead of a single instruction seven would be necessary to compile the and.

```
MX5 1      ⎫ Boolean value
BX5 X5*X0  ⎬ of 1. factor
LX5 1      ⎭ into X5           typtr: boolptr
MX0 1      ⎫ Boolean value     kind:      expr
BX0 -X4*X0 ⎬ of 2. factor      expreg:       0
LX0 1      ⎭ into X0
BX0 X5*X0  and
```

4. The Compilation of Assignments

In addition to the procedure LOAD, a procedure STORE is needed to compile assignments. STORE has two input parameters, namely

the description of the variable to which a new value has to be assigned, and the index of the register containing the new value. The procedure head hence takes the following form:

   procedure store(fattr: attr; fi: regnr)

In analogy to LOAD the global variables ARGS, BRGS, and XRGS are transient to the procedure.

The duality of the load and the store operation implies a certain similarity in the structure of the procedures LOAD and STORE. This manifests itself in a very similar code generation. However, despite these analogies, new problems arise with the compilation of assignments. They all originate from the update of the description of the register contents.

First of all it has to be decided which description should be preferred when alternatives are possible.

Example: i := 1   be compiled to   SX6 B1
                                    SA6 i

Should, in the description of X6, XCONT be set either to SHRTCST (1) or to SIMPVAR (i)? From a theoretical point of view it would be nice if the register description reflected that the constant 1 as well as the actual value of i can be found in X6. However, this would imply that the contents of every register would have to be described by a whole list of entries. Both searching for a particular content and the updating procedure would be more expensive by a factor. For these reasons it was decided not to maintain lists.

When an assignment of the form V := E has to be compiled, three questions are interesting and influence both code generation and the updating of the register descriptions:
- Does a copy instruction (BXi Xfi) have to be generated (fi in [0..5]) or not (fi in [6,7])?
- Is the variable V worth remembering (in the sense of 3.2.1.)?
- Is the expression E worth remembering (in the same sense)?

Depending on the associated answers, the description of the X register contents are updated in STORE as shown in the table below.

E is → V is ↓	a short constant (s)			a long constant (l)			a variable vl worth remembering			another expression		
	fi in [6,7]	fi in [0..5]		fi in [6,7]	fi in [0..5]		fi in [6,7]	fi in [0..5]		fi in [6,7]	fi in [0..5]	
	Xfi	Xi	Xfi	Xfi	Xi	Xfi	Xfi	Xi	Xfi	Xfi	Xi	Xfi
worth remembering	1) v	s	2) v	v	l	v	3) v	vl	v	v	—	2) v
not worth remembering	s	s	s	l	—	2) l	vl	—	2) vl	—	—	—

Remarks:

1) Applied to the above example, this means that the variable i
   is preferred to the short constant 1.
2) And not the inverse as the probability for overwriting Xi
   before Xfi is estimated to be greater than the probability to
   overwrite Xfi before Xi.
3) The "new" variable is preferred to the "old" one.

Evidently, this more or less trivial updating of XRGS is not
sufficient. Through an assignment, certain register content
descriptions can become invalid when they do no longer
correspond to the actual state of the code generation. Three
kinds of assignments must be distinguished:

1. A simple variable (WORDACC = DRCT) is assigned a new value.
2. A simple <u>pointer</u> variable ((WORDACC = DRCT) <u>and</u> (TYPTR↑.FORM=
   POINTER)) gets a new value.
3. An indirect variable (WORDACC = INDCRT) is assigned a new
   value.

For each of these cases an example is given in the table below.

example for case	register content description valid <u>before</u> the assignment	assignment	register content description valid <u>after</u> the assignment
1	args[i]: @v xrgs[i]: v	v := ...	args[i]: @v xrgs[i]: -
2	args[i]: @p↑ xrgs[i]: p↑	p := ...	args[i]: - xrgs[i]: -
3	args[i]: @p↑ xrgs[i]: p↑	p↑ := ...	args[i]: @p↑ xrgs[i]: -

The detection of these obvious coincidences and the
corresponding updating of the register content descriptions
poses no problem. However, things become worse if the same
variable is accessed in two or more different ways. For example,
a variable parameter may coincide with either an actual variable
or another variable parameter. Similarly, two pointer variables
can have the same value whereby the referenced variables
coincide.

There are several ways of overcoming this difficulty.
Sufficient, but not necessary, conditions for the exclusion of
coincidences between two variables are:
- one of them resides in the stack, the other in the heap.
- their types are not compatible.

The former of these criteria is not very selective, while the
latter runs into difficulties in the implementation. For matters
of simplicity, a third solution was implemented. It
distinguishes between two cases, one of them consisting two
subcases:

case 1: A simple variable is assigned a new value.
Coincidence with another simple variable (XCONT = SIMPVAR) can be excluded. Coincidence with a variable parameter, however, is possible. Now, the variable parameters are a subclass of the indirect variables (XCONT = INDVAR). In order to be able to distinguish them from the other indirect variables, the variant SIMPVAR in XRGSTAT has yet another field
                vpaddr: Boolean
telling whether the variable is the address of a variable parameter (VPADDR = TRUE) or not.
Closer inspection of case 1 indicates that it is not necessary to erase the description of all variable parameters in XRGS. It is sufficient to erase those which have a level that is greater than the level of the simple variable. Otherwise coincidence is impossible.

case 2: Another variable is assigned a new value.
Unfortunately, there is no simple criterion to decide whether coincidence with an indirect variable is possible. All INDVAR descriptions in XRGS are therefore deleted. If it is a variable parameter that gets a new value, then the description of all simple variables with a level lower than the level of the variable parameter are also deleted.

Besides the updating of the register content descriptions, another interesting topic comes into view. Should one not - analogously to the load instructions - try to reduce the store instructions? This idea is certainly tempting and was therefore carefully examined during the development of the compiler. An experimental version was built, in which the generation of store instructions was delayed in the case of assignments to SIMPVARs and INDVARs. This could be done by adding a Boolean field to these variants in XRGS telling whether the register contents in question still had to be stored.

It follows that store operations can be eliminated in two cases:

- When a new value is assigned to a variable and the old value has not yet been stored, the generation of the associated store instructions becomes unnecessary.
  For unstructured or unpacked variables this rarely happens. However, with packed structured variables it has some importance, namely when the components of the variables are assigned values one after the other.
  Example (cf. the declarations underlying table 1):

                    with r do
                       begin f1 := true; f2 := ´abcd´ end

- When the end of a procedure body is reached and there are store operations of local variables pending, these store operations are no longer necessary.

Unfortunately, delayed store instructions complicate the compiler considerably and do not bring much in return, if no additional investments are made (as for example removing store operations from loops).

The principal difficulty originates in the fact that the memory is in general not up to date. This implies that special care must be taken when loading a variable. As soon as there is a chance that the variable to be loaded coincides with a register which has yet to be stored, the delayed store operation must take place before the loading.

In addition, it is to be stressed that delayed storing could only be considered for implementation as a compiler option. Otherwise the post-mortem dump facility offered by this implementation would have become a farce as the values printed would never have corresponded to the actual state of the computation.

For these reasons the idea of implementing delayed store operations was finally given up.
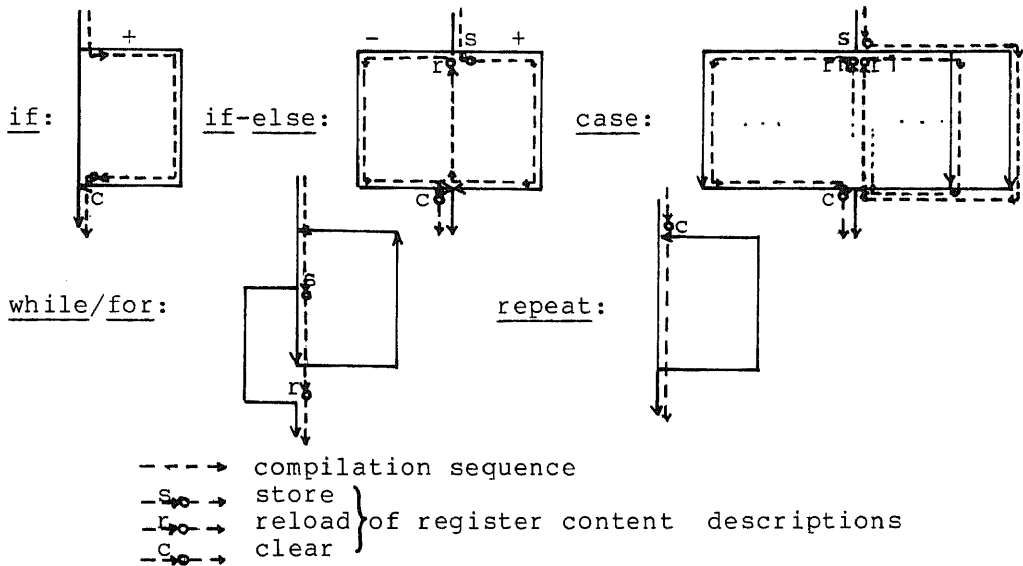
5. The Compilation of the Control Statements

It is characteristic of all control statements that they have a "distributor" D (from which the computation flows out in several directions) and a "concentrator" C (in which the computation flows together from several directions). This applies to
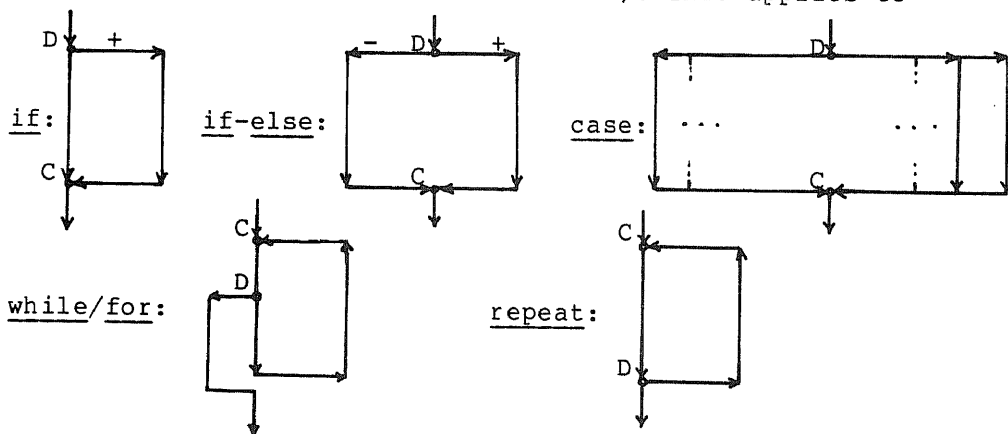
Distributors and concentrators are of great importance to the book-keeping of the register contents. When the compiler reaches a distributor, it must locally store the actually valid register content description in order to be able to reload it when needed. This is necessary because the compilation is strictly sequential, while the execution of these statements is not. Nevertheless the compilation of any stream with distributor D

should profit from the register contents which are valid when the distributor is reached. Therefore the register content descriptions are reloaded in the nodes subsequently marked with an r.

if: if-else: case:

while/for: repeat:

    ---→   compilation sequence
    -s-→   store   ⎫
    -r-→   reload   ⎬ of register content descriptions
    -c-→   clear   ⎭

When the compiler gets to a concentrator, the descriptions of all register contents are cleared since the trade-offs are too heavy to do otherwise:

After compilation of a selective statement, the register content descriptions which are valid at the end of the compilation of every stream would have to be compared to each other. Thereby a description could be derived which would be valid when reaching the concentrator, regardless of how it was reached.

In the case of a repetitive statement the overhead would even be greater. During compilation of the repeated statement, one would be forced to keep track of how code generation depends on the register content description which was valid when the concentrator was reached. This relevant part of the register status would have to be restored at the end of the loop - and this would generally involve code generation.

The compiler procedure WHILESTATEMENT is shown below as an example for the operations of register clear, -store, and -reload.

```
procedure whilestatement;
    var largs: argconts; lbrgs: brgconts; lxrgs: xrgconts;
        lbrg: array [levrange] of regnr; llevels: set of levrange;
        laddr: addrrange; lpl: place;
begin noop; laddr := ic;                                   1)
      clearregs;                                           2)
      expression;                                          3)
      genfalsejump(0); lpl := pl;                          4)
      largs := args; lbrgs := brgs; lxrgs := xrgs;         5)
      lbrg := brg; llevels := levels;
      readnextpascalsymbol;
      statement;                                           6)
      gen30(04b,0,0,laddr);                                7)
      noop; insert(ic,lpl);                                8)
      args := largs; brgs := lbrgs; xrgs := lxrgs;         9)
      brg := lbrg; levels := llevels
end (*whilestatement*)
```

Remarks:

1) The current code word is filled up with NO instructions and the current value of the "instruction" counter IC is saved in LADDR (cf. remark 7) below).
2) The concentrator is reached: the descriptions of the register contents are cleared.
3) The controlling expression is compiled.
4) A false-jump – on the value described by GATTR – is generated. The "pointer" to the involved branch instruction (PL) is saved – thereby allowing a later fix-up of the yet unknown destination address (cf. remark 8) below).
5) The distributor is reached: the descriptions of the register contents are saved.
6) The controlled statement is compiled.
7) The loop is closed: an unconditional jump to LADDR is generated.
8) The current code word is filled up with NO instructions and the branch instruction at LPL is completed with the current "instruction" counter IC.
9) The actually valid register content descriptions are reloaded.

The most elementary control statement is of course the goto-statement. Here is another argument to support those who consider it harmful:

In the terminology introduced in this chapter, labels represent concentrators. These concentrators are, however, far more awkward than the concentrators in "regular" control statements. The reason for this is that, at the labels, computation can flow together from arbitrary directions. Hence, it is impossible, without a complete analysis of the computation flow, to find a nontrivial register content description which is valid when a particular label is reached. Thus, it should not be surprising that in this one-pass compiler the descriptions of the register contents are cleared whenever a label is compiled.

References

[1] Ammann, U., "The Method of Structured Programming Applied to the Development of a Compiler", International Computing Symposium 1973,93-99, A. Guenther et al. (eds.), North-Holland (1974)

[2] Ammann, U., "Die Entwicklung eines PASCAL-Compilers nach der Methode des Strukturierten Programmierens", Juris-Verlag Zürich (1975)

[3] Jensen, K. and Wirth, N., "PASCAL - User Manual and Report", Lecture Notes in Computer Science, No. 18, Springer-Verlag, Berlin-Heidelberg-New York (1974)

[4] Hoare, C.A.R. and Wirth, N., "An Axiomatic Definition of the Programming Language PASCAL", Acta Informatica 2, 1973, 335-355

[5] Nori, K.V. et al., "The PASCAL P-code Compiler: Implementation Notes", ETH Zürich, Berichte des Instituts für Informatik, No. 10, 1975

[6] Knuth, D.E., "The Art of Computer Programming", 1, Addison-Wesley, 1968

[7] Wirth, N., "The Programming Language PASCAL", Acta Informatica 1, 1971, 35-63

[8] Wirth, N., "The Design of a PASCAL Compiler", Software - Practice and Experience, 1, 4 (1971), 309-334

[9] Control Data 6400/6500/6600 Computer Systems Reference Manual

Appendix: An Example of the Code Generation

Knuth [6] gives an algorithm to compute the date of Easter. The following implementation of this algorithm is intended to facilitate insight into the code generated by the new PASCAL compiler.

```
          program easter(output);
            const lim1 = 1976; lim2 = 2000;
            type year = lim1..lim2; month = 3..4; day = integer;
            var ye: year; mo: month; da: day;
            procedure dateofeaster(y:year; var n:day; var m:month);
              var g,c,x,z,d,e: integer;
000002    begin
000010      g := y mod 19+1;
000013      c := y div 100+1;
000016      x := 3*c div 4-12;
000021      z := (8*c+5) div 25-5;
000026      d := 5*y div 4-x-10;
000032      e := (11*g+20+z-x) mod 30;
000040      if e < 0 then e := e+30;
000042      if (e=25) and (g>11) or (e=24) then e := e+1;
000052      n := 44-e;
000054      if n < 21 then n := n+30;
000057      n := n+7 - (d+n) mod 7;
000065      if n > 31 then
000067          begin n := n-31;
000067                m := 4
000070          end
000071      else m := 3
000072    end(*dateofeaster*);
000002  begin
000021    for ye := lim1 to lim2 do
000024      begin dateofeaster(ye,da,mo);
000026          (*output of ye,da,and mo here*)
000026      end (*for*)
000032  end.
```

The reasons for choosing this program are manifold. It contains a procedure with value and variable parameters, a for-loop, several conditional statements, and nontrivial arithmetic.

The addresses associated with the variables occurring in this program are:

Variable:	Address:	
ye	B2+151	⎫
mo	B2+152	⎬ main program variables
da	B2+153	⎭
y	B5+3	⎫
n	B5+4	⎬ parameter descriptors
m	B5+5	⎭
e	B5+6	⎫
d	B5+7	⎪
z	B5+8	⎬ variables local to
x	B5+9	⎪ DATEOFEASTER
c	B5+10	⎪
g	B5+11	⎭

The compiler's internal description of the register contents during compilation of the procedure DATEOFEASTER is summarized below. Each line shows the register contents between two statements.

program address	X registers: X0 X1 X2 X3 X4 X5 X6 X7	A registers: A1 A2 A3 A4 A5 A6 A7
000002	y @n @m	
000010	y @n @m	@m
000013	y @n @m 19 1 g	g
000016	y @n @m 100 1 g c	g c
000021	y @n @m 12 1 x c	x c
000026	y @n @m 25 5 z c	z c
000032	y n @m x 10 z d	x z d
000040	g 30 x z e	g x z e
000042	0 g 30 x e e	g x e e
000052	25 1 g 11 24 e	e g e
000054	44 @n e n	@n n
000057	44 @n e 21 30 n	@n n
000065	7 @n d n	@n n d n
000067	7 @n 31 n d	@n n d n
000067	7 @n 31 d n	@n n d n
000071	7 @n 31 @m d n m	@n n @m d n m
000072	7 @n 31 n d @m m	@n n d @m m

With the help of this information, the compiler generates the following code:

DATEOFEASTER: Remarks:

```
000002 SX6 B5      ⎫   procedure entry code
       LX6 18      ⎪
       BX7 X7+X6   ⎪     1) stores dynamic link (B5) and return
       SB5 B6      ⎬        address (X7); the static link is not
                   ⎪ 1)    used and therefore not set. B5 (BASE)
                   ⎪       and B6 (NEXT) are updated.
000003 SA7 B5+B1   ⎭
                   ⋮
```

```
          SB6  B6+12    ⎤       2) Tests whether stack and heap risk
          NO             ⎦          overlapping (error address in A0).
                                 3) Saves the parameter descriptors
000004    SB7  B6+100   ⎞           passed in X registers, i.e. the
          SA0  000005   ⎰ 2)        value of y (in X0) and the addresses
                                    of n and m (in X1 and X2 respectively).
000005    GE   B7,B4,000051  ⎤
          BX6  X0              ⎮
          NO                  ⎮
                              ⎮
000006    SA6  B5+3           ⎬ 3)
          BX6  X1             ⎮
          SA6  A6+B1          ⎮
                              ⎮
000007    BX6  X2             ⎮
          SA6  A6+B1          ⎦


          SX3  19                  g := y mod 19+1

000010    PX5  X0     1)  ⎤       1) The value of y is already in X0.
          PX4  X3         ⎮       2) This is the CDC 6000 standard
          NX4  X4         ⎮          instruction sequence for integer
          FX4  X5/X4      ⎬ 2)       divisions [9].
                          ⎮       3) y mod 19 = y-y div 19*19
000011    UX4  B3,X4      ⎮       4) makes use of B1 = 1.
          LX4  B3,X4      ⎦
          DX4  X3*X4      ⎫
          IX4  X0-X4      ⎬ 3)

000012    SX5  B1     4)
          IX6  X4+X5
          SA6  B5+11
            .
            .
            .
000032    SX3  11
          SA1  B5+11                e := (11*g+20+z-x) mod 30

000033    SX2  X3*X1                1) The value of z is already in X6.
          SX5  20                   2) The value of x is already in X4.
          IX2  X2+X5                3) The hidden multiplication by 30
                                       is optimized: 30=2**1*(2**4-1).
000034    IX2  X2+X6   1)           4) The address of d , which is
          IX2  X2-X4   2)              adjacent to the address of e ,
          SX3  30                      is already in A7.

000035    PX0  X2
          PX5  X3
          NX5  X5
          FX5  X0/X5

000036    UX5  B3,X5
          LX5  B3,X5
          IX0  X5+X5
            .
            .
```

```
        BX5 X0
000037  LX5 4              }  3)
        IX5 X5-X0
        IX7 X2-X5
        SA7 A7-B1             4)
        .
        .
        .
```

if n > 31 then
begin n := n - 31; m := 4 end

```
        BX3 X6             1)
000065  SX2 31
        IX5 X2-X6             2)
        NO
000066  PL  X5,000070
        IX6 X3-X2            3)
        SA6 A2              4)
000067  SA3 A1+B1           5)
        SX7 4
        SA7 X3
000070  EQ  000072
        .
        .
000072  SA1 B5+B1
        SB6 B5
        SB7 X1
        LX1 42
000073  SB5 X1
        JP  B7
```

1) Copying one of the bottleneck
   registers X6/X7 is expected to
   be more fruitful than generating
   a NO instruction.
2) The value of n is already in X6.
3) 31 was found in X2, the value of n
   was found in X3.
4) The address of n is already in A2.
5) An address adjacent to the one of m
   was found in A1.

procedure exit code

   restore B5 and B6 and return

EASTER:
```
        .
        .
        .
000021  SX6 1976
        SX0 2000
000022  IX0 X0-X6           1)
        NG  X0,000030       2)
        BX0 X6
000023  SA6 B2+151
```

for ye := lim1 to lim2 do

1) Here the loop is closed. The in-
   variant is: X0 contains the value of
   the upper limit and X6 contains the
   "new" value of the control variable.
2) Again, to avoid a NO instruction.

dateofeaster(ye,da,mo)

```
                            1)
        SX1 B2+153
                            2)
```

1) The value of ye was found in X0.
2) Up to 5 parameter descriptors are

```
000024 SX2 B2+152              passed in X registers (X0-X4). This
       SX7 000026              shortens the calling sequence and
                               allows starting the compilation of
000025 EQ  DATEOFEASTER        many procedures with a nonempty
       NO                      register status.
       NO


000026 SA1 B2+151          end (*for*)
       SX0 B1
       IX6 X1+X0               The invariant is established
                                and the loop is closed.
000027 SX0 2000
       EQ  000022

000030
```

In the whole of this example the compiler uses 12 times the fact
that the value of a simple variable is already in an X register;
and 5 times, it finds the value of a variable parameter in a
register. In addition, it realizes 3 times that the address of a
variable parameter (i.e. the value of its descriptor) happens to
be in a register. Hence, the number of load instructions is
reduced from 37 to 12, thereby saving 25 (=12+5*2+3) - i.e. a
saving of more than 2/3.

Several of the numerous occurrences of a constant give rise to
improved code generation. Five multiplications (3*c, 8*c, 5*y,
and the two hidden ones in ... mod 30 and ... mod 7) are
compiled to shifts and additions, and for two divisions (both of
the form ... div 4) shift instructions are generated.

In fifteen places a 15 bit instruction can be generated instead
of a 30 bit instruction. Two times the address of a local
variable and ten times a neighbouring address is found to be in
a register. Three times the compiler can generate code to load
the constant 1 from B1.

To compare the efficiency and the tightness of the code produced
by the new compiler this sample program was also input to the
old PASCAL compiler. It turned out that the code generated by
the old compiler was about 30 % less tight and about 25 % slower
in execution.

Berichte des Instituts für Informatik