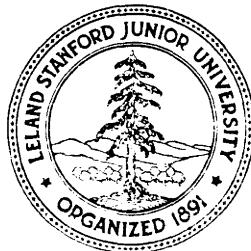# ON "PASCAL", CODE GENERATION, AND THE CDC 6000 COMPUTER

BY

NIKLAUS WIRTH

STAN-CS-72-257
FEBRUARY 1972

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

O. r "PASCAL" Code Generation, and the CDC 6000 Computer

by Niklaus Wirth

Abstract:

"PASCAL" is a general purpose programming language with character-istics similar to ALGOL 60, but with an enriched set of program- and data structuring facilities. It has been implemented on the CDC 6000 computer This paper discusses selected topics of code generation, in particular the selection of instruction sequences to represent, simple operations on arithmetic, Boolean, and powerset operands. Methods to -implement recursive procedures are briefly described, and it is hinted that the more sophisticated solutions are not necessarily also the best. The CDC 6000 architecture appears as a frequent source of pitfalls and nuisances, and its main trouble spots are scrutinized and discussed.

On **"PASCAL"**, Code Generation, and the CDC 6000 Computer

1. <u>Introduction</u>

This set of notes has a dual purpose. It is on the one hand
directed to the <u>user of the PASCAL compiler</u> system who would like to
gain some insight into the machine code which is generated for various
basic operations. It is even recommended that he study these notes
carefully, because their understanding may prevent him from certain
pitfalls which are inherent in the use of the CDC 6000 computer [1].

On the other hand the notes-may be of interest to <u>compiler writers</u>
in general, because they point out some problems and dilemmas and our
choices of solutions. It becomes apparent that the choice of the code
to be generated is crucial for a good compiler system, and that it is
far **from** trivial as is usually believed.

The true purpose of a higher-level language is that it allows a
programmer to conceive his algorithms in terms of some <u>convenient</u>
<u>abstractions</u>. For instance, he is given the opportunity to think in
terms of familiar notions of numbers, of relations, and of repetitions,
instead of having to express his program in terms of bitstrings,
arithmetic instructions, and transfers of control. However, these
abstractions are only truly useful, if he can assume that his implemen-
tation observes all the <u>properties</u> which are <u>commonly attributed</u> to
these abstractions, or else if it <u>automatically issues a warning</u>. As
an example, when dealing with numbers in a high-level language, one
should like to assume all the **common axioms** of arithmetic to hold. Of
course this is not possible, since computers can only represent <u>finite</u>
ranges of values. So one expects to receive a warning, if an operation

1

has trespassed the limits imposed by the implementation and an operation generates a result not in accord with the rules governing the abstraction. So the system is expected to provide an error indication, e.g. if an overflow occurs in an addition, if a value is being assigned which lies outside the specified range of values of variables, or if an array index is used which lies outside the defined limits.

Unfortunately, such potential warnings require the execution of additional instructions, which in general is costly. As far as range checking is concerned, they can be requested to be generated by the compiler for run-time execution by enabling so-called options. (The A-option generates assignment range checks, the X-option index checks.) They are relatively costly, but may speed up the finding of logical mistakes a great deal.

As far as irregularities of the arithmetic are concerned, one has become used to receive these warning signals automatically from the hardware, particularly because they are easily generated by the hardware, whereas a solution to detect overflow by software is usually beyond any reasonably economical feasibility. Unfortunately, the CDC computer fails to satisfy even the most modest expectations in this respect, and the effort to provide a system with security in the above sense was therefore a series of constant frustrations. Equally disappointing are some of the "features" of its floating-point arithmetic instructions.

One can go only a relatively short distance in trying to correct mistakes of the hardware by means of software; otherwise a system becomes ridiculously inefficient and will not be used by conscientious programmers who are willing to take the peculiarities of a hardware into account and guarantee safety of their algorithms by analytical

rather than experimental means. And this would have been against the intentions of PASCAL. So all that can reasonably be done is to elucidate the shortcomings and limitations of the hardware that are still transparent through the "software cover", and to make the programmer fully aware of them. And this is the purpose of this note.

It concerns itself with the simple operations of integer and real arithmetic, with Boolean operations and with powersets. The reader is supposed to be familiar with the CDC COMPASS notation. The operands are usually assumed to have been brought into the X1 and X2 registers. (If they were loaded into other registers, a corresponding renumbering is necessary which is, however, irrelevant to the operation itself). Registers X1-X5 are used as a stack for intermediate results, whereas X0 is used exclusively as local work register.

Section 6 deals with the topic of implementing recursive procedures and the addressing of local variables. Although the general techniques are well-known, analysis of possible solutions and their experimental **comparison** yielded some noteworthy results. It is shown that attempts to make full use of available hardware features such as base registers may not necessarily lead to an optimal performance. Again, the instruction set of the CDC computer is hardly optimal to implement mechanisms for recursive procedures. Conspicuously absent is a sub-routine jump instruction which leaves the code invariant (reentrant).

2. Integer Arithmetic

Data of type integer or of subranges thereof are represented by fixed-point binary numbers. Addition and subtraction are represented by the

IXi   Xj $\pm$ Xk

instructions. Other operations are implemented by short sequences of instructions, as outlined below.

2.1 **Multiplication**

Due to a recent change of the hardware, fixed-point multiplication can be performed by a single

DXl   x1*x2

instruction. It should, however, be noted that this instruction is essentially a floating-point instruction, and yields incorrect answers for fixed-point operands with $|x| \geq 2^{48}$. This can be regarded as an overflow condition which is, alas, neither trapped nor indicated by the computer. A "safe code", checking against all imposed limits of operands and result, is quite elaborate and uneconomical by any standards, and was therefore not implemented.

If one of the operands is a constant C being representable as either

1. $c = 2^n$ (2, 4, 8, 16 ...)

2. $c = 2^m + 2^n$ (3, 5, 6, 9, 10, 12 .1

                 $m > n$

3. $c = 2^m - 2^n$ (7, 14, 15 ...)

then the compiler generates the following code for the multiplication
of X1 by c :

```
    1:   LX1   n              multiply by 2"
  2,3:   LX1   n
         Bxo   X1
         LX1   m−n
         IX1   X1+XO
```

Again, overflow conditions are simply ignored. Case 3 yields only
correct results, if $|X1*2^m| < 2^{59}$.

2.2 Division (div)

Integer division is represented by the instruction sequence

```
    PX1   X1   ⎫
    PX2   x2   ⎬ pack
    NX2   x2   ⎭
    FX1   X1/X2    divide
    UX1   B7,X1
    LX1   B7,X1
    BXO   XO−XO  ⎫ suppress neg. zero
    1x1   X1+XO  ⎭
```

and suffers from the same basic shortcoming as multiplication: an
operand $|x| \geq 2^{48}$ yields an incorrect result.

If the divisor is a constant $c = 2^n$, the compiler again produces
an "optimized" code, performing division by shifting. Unfortunately,
a single right shift instruction is unsatisfactory, because it may
generate a "negative" zero as result. Negative zeroes, however, must
not be allowed to occur, since comparisons may yield wrong answers if
applied to them. Thus, the optimized division is implemented as

```
    AX1   n          divide by 2^n
    BXO   XO−XO      suppress
    IX1   X1+XO      negative zero
```

Note that the unconditional generation and addition of a zero can be

5

accomplished with a code that is not only shorter than a conditional
jump, such as

```
        AX1  n
        NZ   X1,L
        SX1  BO
    L   ...
```

but also avoids the insertion of padding instructions (NOPs) for word
boundary alignment.

The D-option provides an additional security measure against
division by zero. It causes the compiler to insert a

```
        ZR   X2, error
```

jump instruction preceding every division instruction. (This applies
to the Modulus operation as well.) It is particularly recommended in
the case of integer division, where the actual divide instruction
generates a "floating-point infinity" value, which is incorrectly
treated by the subsequent conversion instructions and thereby represents
a senseless result.

2.3 Modulus (mod)

The modulus or remainder operation is defined as

$$x \bmod y = x-(x \text{ div } y) * y$$

As it involves integer multiplication and division operations, it
suffers again from the same deficiencies of the 6000 arithmetic. Its
corresponding code is:

```
        PXD  X1
        PX6  x2
        NX6  X6
        FX6  X0/X6
        UX6  B7,X6
        LX6  B7,X6
        DX6  X6*X2
        IX1  X1-X6
```

2.4 Sign inversion

The use of **one's** complement representation for negative numbers
makes again the most obvious choice of code

        BX1   -X1

unsatisfactory, because it might generate a "negative zero". So we use

        BX0   X0-X0
        IX1   X0-X1

2.5 Comparisons

Since the computer does not offer a compare instruction, subtraction
has to be used; this has primarily the disadvantage of generating wrong
results in the case of overflow. The cases of testing for equality and
inequality are handled correctly, because the **one's** complement addition
generates an end-around-carry in the case of "negative overflow", thus
maintaining a result indicating inequality. Note that the Boolean
subtraction

        BX1   X1-X2

cannot be used, because a comparison of x1 and x2 = -x1 would yield
a zero result, thus indicating equality.

Whereas equality testing is **"safe"** with the

        IX1   X1-X2

instruction ignoring overflows, this is not the case for the tests of
ordering (x1 < x2) by subtraction and subsequent inspection of the
sign bit. The reason is that if overflow occurs, i.e., $|x1 - x2| > 2^{59}$,
then the sign bit will be the opposite of the true sign. This situation
is quite hopeless, since overflow is in no simple way detectable on this
machine. In order to obtain a (sign) bit representing the relation

x < y for any values x,y , the following algorithm can be used:

1. Compare the signs of the two operands.

2. If they are different, then the result is obvious.

3. If they are equal, the subtraction x-y can be performed

without danger of overflow, and x-y < 0 is the result.

A minimal instruction sequence to perform these operations and avoiding

the use of undesirable jump instructions is

```
BX0   X1-X2       compare sign bits
IX2   X1-X2
BX1   X0*X1       if unequal, choose sign of X1
BX2   -X0*X2      if equal, choose sign of X1-X2
BX1   x1+x2
```

Now the sign bit of X1 is 1, if X1 <X2 , and 0 otherwise. Still,

the effort to perform a faultless comparison is formidably cumbersome,

and the PASCAL compiler does not generate it. The programmer is left

with the responsibility to verify that for every comparison of x and y ,

$|X-Y| < 2^{59}$.

2.6 Taking the absolute value (ABS)

The code used to take an absolute value is designed to avoid jump

instructions, not only because they are long and slow, but because they

usually introduce NOP instructions for alignment.

```
Bxo  X1  .
AX0   59        generate 60 sign bits
BX1   X0-X1
```


2.7 Testing for even or odd (ODD)

Since one's complement representation is used for negative numbers,

the least significant bit of the operand must be compared with its sign

bit:

```
            BXO   Xl
            LXO   59
            BXl   Xl-XO
```

This leaves the sign-bit of Xl equal to 1 , if Xl was odd, and

0 otherwise.   ___

The compiler "optimizes" in the case of ODD(x) with x being of

a **subrange** type with only non-negative values.   It then generates the

single instruction

```
            LXl   59
```

2.8 Summary

The foregoing explanations reveal that the absence of any overflow

indication makes analytical verifications necessary that guarantee the

non-occurrence of these conditions. An effective aid in experimental

testing is the A-option, causing interval check instructions to be

generated with every assignment to a variable that is declared to be of

a subrange'type. The A-option is activated by the "comment"

```
            {$A+ ... }
```

and causes the code for an assignment to a variable

```
            VAR V: a..b
```

to become:

```
            SX7   *              location identification for error trap
            SXl   a
            1x0   X6-Xl
            SXl   b
            IXl   Xl-X6
            BXO   Xl+XO
            NG    XO,error        jump to error routine
            SA6   V
```

It should be noticed that unfortunately the attractive and shorter code

sequence

```
sx7    *
sxo    x6-a
SX1    x6-b-1
BXO    -X1+XO
NG     XO, error
SA6    v
```

cannot be used, because the instructions

$$SXi \quad Xj \pm K$$

perform an 18-bit arithmetic ignoring the leading 42 bits of the

register Xj which -- of course -- is not in the spirit of a check.

This ignoring rather than checking of the leading bits in 18-bit

arithmetic is the reason why the so-called "increment" instructions

cannot be used by the PASCAL compiler, except in the following special

circumstance: if a variable x is declared of a subrange whose limits

are both less than 2^{17} in absolute value, then the assignment statement

$$x := x + k$$

is compiled as

```
SA1    x
SX6    X1+k
SA6    x
```

3. Floating-point Arithmetic

The PASCAL compiler uses the canplete set of F-instructions for arithmetic with values of type "real". Comparison is performed by subtraction due to the lack of a **compare** instruction. This is possible without handicap since the occurrence of overflow generates a <u>signed</u> "infinity" -value, but no immediate trap. Sign inversion is represented by

```
        BX0   X0-X0     generate zero
        IX1   X0-X1
```

and the <u>absolute value</u> function by

```
        BX0   X1
        AX0   59
        BX1   X0-X1
```

Arithmetic with the F-instruction possesses some peculiar properties which will briefly be reviewed, and has for instance the consequence that $x-y = 0$ does not necessarily imply $x = y$, if the difference is computed by an **Finstruction.** The trouble arises from the fact that F-arithmetic truncates without rounding, and F-addition truncates without post-normalization. Every addition is therefore compiled into two instructions:

```
        FX1   X1 + X2     add/subtract
        Nx1   X1 ‾        post-normalize
```

If the two values

$$a = 1720\ 40...00B \qquad = 1.0$$
$$b = 1717\ 17...77B \qquad = 1.0-2^{-48}$$

are compared by subtraction

```
        FX0   X1-X2       a-b
```

the result is

$$1720\ 40\ldots\quad \ldots00\ /\ 00\ldots\quad \ldots\circ$$
$$-1720\ 37\ldots\quad \bullet\quad ..77\,/\,40\ldots\quad \ldots\,\circ$$
$$1720\ \circ\circ\ldots\quad \ldots00\ /\ 40\ldots\quad \ldots\,0$$

where the slash marks the separation between the lower and the upper
half of the 96-bit accumulator. The result is 0 although the two
operands were different.

Notice that subtracting 0.5 from both a and b , and then
computing their difference, yields

$$a\ -\ 0.5 = 0.5 \qquad :\ 1717\ 40\ldots\ \ldots\ 00$$

$$b\ -\ 0.5\ =\ 0.5\text{-}2\text{-}48\ \ :\ \ 1716\ 77\ldots\ \ldots\ 76$$

$$1717\quad 40\ldots\quad \ldots\circ\ 0\,/\,00\ \ldots\quad \ldots\ 0$$
$$-1717\quad 37\ldots\quad \ldots77\,/\,00\ \ldots\quad \ldots\ 0$$
$$1717\quad 00\ldots\quad \ldots01\,/\,00\ \ldots\quad \ldots\ 0$$

i.e., a difference which is not zero. Thus the result does not only
depend on the true result, but also on the values of the operands.
This unpleasant, property of the CDC F-arithmetic stems from the fact
that automatic post-normalization is absent.

3.1 Rounding

It was at one time hoped that this defect could be avoided by
letting the PASCAL compiler automatically generate R-instructions,
which include a certain kind of rounding. However, R-arithmetic
turned out to feature some even stranger properties, so that it was
decided not to use R-instructions. In order to point these features
cut, a brief review over R-arithmetic is necessary:

The R-instructions differ from the F-instructions only insofar as a 1-bit is appended to normalized operands before the arithmetic operation is performed. Thus for instance the subtraction of $b = 1.0-2^{-48}$ from a = 1.0 yields

$$
\begin{array}{llll}
1720 & 40\ldots \quad \ldots 00 \;/\; & 40 \ldots \quad \ldots 0 \\
-1720\underline{\quad 37\ldots \quad \ldots 77\;}/\;\underline{\; 60 \ldots \quad \ldots 0\;} \\
1720 & \underline{\;0\,0\ldots \quad \ldots 00\;} \;/\; & 60 \ldots \quad \ldots 0
\end{array}\;\Big\}\; \text{1-bits appended}
$$

which of course is still zero.

The principal defect with "CDC-rounding", however, is that its effect is unpredictably either the addition of 1/2 or 1/4 in the last position, because rounding takes place _before_ instead of after normalization (which must again be performed by a separate instruction). The following example illustrates this, which is shown on hand of a five-bit number representation:

$$
\begin{array}{rll}
16 = & 10000 \;/\; 1 & \text{inserted} \\
+17 = & 10001 \;/\; 1 & \text{round-bits} \\
33 = & 100010 \;/\; 0 \\
& \underline{10001} \;/\; 0 \;= 34
\end{array}
$$

$$
\begin{array}{rll}
31 = & 11111 \;/\; 1 & \longleftarrow \text{inserted round-bit} \\
+2 = & 00010 \;/\; 0 \\
33 = & 100001 \;/\; 1 \\
& \underline{10000} \;/\; 1 \;= 32
\end{array}
$$

In the first case, the pre-rounding results in correct rounding of the not exactly representable 33 to 34 , whereas in the second case pre-rounding has no effect.

The same phenomenon can be observed in the cases of multiplication and division. The following example again uses a five-bit number representation:

13

```
                    round-bit
                        1
   15 x 12  =     11110 / 1  x    11000
                  11110 / 1 ◄──────┐
               +  01111 / 01 ◄─────┘
                 101101 / 11
                 10110 / 111                = 176
```

```
                    round-bit
                        1
   18 x 10  =     10010 / 1  x    10100
                  10010 / 1 ◄──────┐
               +   100 / 101 1 ◄───┘
                  10111 / 001               = 184
```

In the first case, the rounding effect is nil, leaving the inexactly
representable value 180 be an unrounded 176; in the latter case the
rounding effect transforms 180 into the value 184. (Suitable adjust-
ment of exponents is not shown here.)

A method introducing proper rounding instead of "CDC-rounding"
relies on the use of the D-instruction set [2]. Whereas the F-instructions
yield the high-order 48 bits of the 96-bit accumulator, the D-instructions
yield the low-order 48 bits with a suitably adjusted exponent, thereby
allowing access to a double precision result.

Notice that it is an ingeniously efficient method to compute a
double precision result by

1. computing the DP-result and dispose of the low half
    (F-instruction), then

2. computing the same again and dispose of the high half
    (D-instruction).

This computer allows it to be done in no other way!

The PASCAL compiler will generate the following code for **floating-point** operations, depending on the choice of the <u>R-option</u>:

R-option	OFF		ON	
x ± y	FX1	x1+ x2	FXO	x1+ x2
	NX1	X1	Nxo	XO
			DX1	x1+ x2
			RX1	X1 ∓ XO
			NX1	X1
x * y	FX1	X1 * x2	FXO	x1* x2
			DX1	x1* x2
			RX1	x1+ XO
x / y	FX1	X1 / x2	RX1	X1 / x2

Examples of addition/ subtraction:

1.
$$1.0$$
$$1.0-2^{-48}$$

1720	40...	...00 / 00...		O
1717	77...	...77 / 00...	:::	O
1720	37...	...77 / 40...	...	O

1720	00...	...00 / 40...	...	O

= 1640 40... . ..00   after addition of high
                              and low

2. Take a = 1.0 and b = 2^{-48} , then subtract a-b :

F-subtraction yields

a =   1720 40... ...00 / 00... ... O
b =   1720 00... ...00 / 40... ... O
     ─────────────────────────────────────
      1720 37... ...77 / 40... .

which, after normalization, is

<u>1717</u> 77... · ..76 = 1.0-2^{-47}

R-subtraction inserts a 1-bit after the slash in the first operand, and thus yields the result

<u>1720</u> 40... ...00 = 1.0 exactly

The combined use of F and D instructions yields the true result, because the normalization instruction left shifts the high order result to

$$1717 \quad 77\ldots \quad \ldots 76$$

whereafter a "rounded" -addition is used to add the correction

$$+ \: 1717 \quad 009.. \quad . \, ..01$$

yielding

$$\underline{1717 \quad 77\ldots \quad \ldots 77} \: = \: 1.0\text{-}2^{-48} \quad .$$

3.2 Conversion from fixed to floating-point (integer to real)

Wherever a real operand is permissible, PASCAL allows the specification of an operand of type integer as well. However, the compiler is theh forced to generate the necessary representation conversion instructions, which are not only time-consuming, but potentially hazardous. It is therefore recommended to avoid "mixed-mode" arithmetic expressions wherever possible. The generated conversion instructions are

```
PX1    BO,X1    pack with zero exponent
NX1    BO,X1    normalize
```

The result of this conversion is wrong, whenever the integer operand in X1 is larger or equal to 2^{48} in absolute value, since the exponent bits are simply ignored by the P instruction. A test to verify that the operand is within bounds could be compiled as

```
Bxo    X1
AXO    48
NZ     XO, error
```

but is easily seen to be more costly than the conversion itself.

16

3.3 Conversion from floating to fixed-point (real to integer)

PASCAL does not provide for any implicit real to integer conversion. However, the standard **function TRUNC**(x) allows to truncate the fractional part of a real number. The used code is:

```
UX1    B7,X1
LX1    B7,X1
BX0    X0-X0   ⎫ avoid
IX1    X1+X0   ⎭ negative zero
```

The result of this conversion is again wrong, if $|x| \geq 2^{48}$.

4. Boolean Operations

The standard type Boolean is defined in PASCAL as

<u>type</u> Boolean = (false, true)

Since the values of all scalar types are mapped onto the integers

0,1,2,... , the values <u>false</u> and <u>true</u> are represented by the numbers

0 and 1 respectively.

The operations $\wedge$ and $\vee$ are implemented by the Boolean AND and

OR instruction, namely

```
BX1    x1*x2    and
BX1    x1+x2
```

<u>Negation</u> is performed by

```
MX0    59
BX1    -X0-X1
```

If a relation has to be assigned to a Boolean variable, e.g.

b : = x < y

then a sequence of instructions is necessary to obtain a 0 or 1

value. Again every effort is made to avoid the use of jumps. The

following code is used in the above assignment; leaving a Boolean value

in X1 .

```
FX1    X1-X2    x-y
Mxo    1
BX1    X0*X1    Extract sign bit
LX1    1        move it to correct position
```

Analogous code is generated for the relations > , $\leq$, and $\geq$. But

unfortunately the equality relations cannot be reasonably implemented

without a jump; in the assignment

b := x = y

the following instructions are generated:

```
      FXO   X1-X2
      BX1   X0-X0
      NZ    X1,L
      SX1   1
  L ...
```

Boolean comparisons, although occurring rather infrequently, are treated

as special cases, because a simpler and shorter code is applicable:

$$p < q \qquad BX1 \quad -X1*X2$$

$$p \leq q \qquad \begin{matrix} BX1 & -X2*X1 \\ MX0 & 59 \\ BX1 & -X0-X1 \end{matrix} \Bigr\} \text{ negation}$$

$$p \neq q \qquad BX1 \quad X1-X2$$

The remaining three relations are **compiled** analogously.

5. **Powerset** Operations

PASCAL 6000 restricts **powerset** types to be built only on base sets with less than 59 components. This allows a **powerset** value S to be represented by one "word", in which the i-th bit indicates the presence (1) or absence (0) of the element i in S .

5.1 Generation of the Singleton Set [i]

Assume that i is loaded into register X1, then

```
SB7    X1
SX1    1
LX1    B7,X1
```

Notice that the numbering of bits starts with 0 at the low order end. This choice was made in order to be able to load **powerset** constants with small valued components (less than 18) by a single **SXi** instruction.

5.2 Set Intersection, Union, and Difference

These three operations are implemented by a single instruction

intersection	**BX1**	x1*x2
union	**BX1**	x1+x2
difference	**BX1**	-X2*X1

5.3 Set Membership (**in**)

The relation i in S is implemented by shifting the bit representing i into the sign position which can be tested:

```
SB7     X1    i
AX1     B7,X2    S
LX1     59
```

If the expression i is in the form of a constant c , then the compiler generates of course only the single instruction

```
LX1    59-c
```

5.4 Set Comparison

Sets can be compared for equality and inclusion. <u>Equality</u> is tested by a Boolean subtraction

        BXO   X1-X2

and a subsequent zero test. Note that the peculiar property of the zero test to recognize a word with either 60 zero-bits or 60 one-bits as a zero is responsible for the restriction that powersets may contain at most 59 instead of 60 elements. If sets with 60 components were allowed, then a full set and an empty set would not be distinguishable by a single subtraction followed by a zero-test.

<u>Inclusion</u> expressed as $x \leq y$ and meaning $x \subseteq y$, is implemented by the single instruction

        BXO   -X1*X2

which is followed by a zero-test instruction. The **same** instruction is used for the relation $x > y$, whereas strict inclusion $(x \subset y)$ is not implemented.

Some Exercises Addressed to the CDC 6000 Expert

1. Is the following code to represent the function trunc(X1)

    acceptable? If not, why?

        BX0   X0-X0
        PX0   X0
        FX1   X1+X0
        UX1   B7,X1
        NZ   B7, overflow

2. Is the following code for X1 mod X2 acceptable? If so, prove it.

        PX1   X1
        Px2   x2
        NX6   x2
        FX6   X1/X6
        Bxo   X0
        FX6   X6+X0
        DX6   X6*X2
        FX6   X1-X6
        UX1   X6

3. Why can the instructions

        BX0   X1-X2
        ZR   X0, equal

    not be used to represent a comparison X1 = X2 ? Prove that

        1x0   X1-X2
        ZR   X0, equal

    always yields the correct action.

6. <u>Irnplemcntution of Recursive Procedures</u>

The language PASCAL has been carefully designed so that dynamic storage allocation is not required, with the following two exceptions:

1. Variables local to procedures may be allocated storage only when the procedure is called, and

2. Components of class variables are allocated storage by calling the standard procedure "alloc". An area of store is allocated to the entire class variable as soon as the procedure is called to which the class is local.

In this section we will briefly review the well-known techniques for handling recursive procedure calls and of allocating storage to their local quantities, and discuss the code selected to represent the procedure call mechanism.

Due to the first-in last-out nature of the hierarchy of activated procedures a stack may be used to allocate local variables. This is of great advantage, since storage retrieval is trivial in the case of stacks, resulting in low storage management overhead. We consider the set of local variables of each activated procedure as a record (often called "data segment") in the stack. Since their lengths may all be different, the most convenient method to thread the way back through such a stack is by constructing a chain of pointers linking the records. Every record then contains a "header" containing

1. the link to the previous record, and

2. the (frozen) program status (counter) of the calling procedure.

Variables are addressed relative to the origin of the record of which they are a part. The origin address is unknown at compile-time,

and must be determined at run-time. This can be done by descending through the link chain, until the desired record is reached. But how is the desired record recognized'? The most straight-forward method which interprets the scope rules of an ALGOL block structure correctly is probably the following:

Method I:

1. Define the level of an object to be 1 greater than the level of the procedure to which it is local. The level of the main program is 0 .

2. Indicate the level of each record (equal to the level of its components) in its heading.

3. Whenever an object on level i has to be accessed, the record containing it is found by descending down the chain of links until the first occurrence of a level indicator with value i is found.

This accessing method has the obvious drawback of inefficiency (and of not being applicable in the case of parametric procedures). A slight modification, however, improves efficiency and generalizes to parametric procedures.

Method II:

Instead of indicating levels explicitly in the record headings, a second link chain is constructed connecting each record A with its static ancestor, i.e., with the record B of the procedure in which A was declared locally. In order to distinguish the two link chains, the former is called the "dynamic link" and the latter the "static link". An example of a state of computation is shown below for a given -- admittedly not very realistic -- program.

```
var v0;

procedure Q0 (procedure X);

    var w1;

    procedure Q1;

        var w2;

    begin w2 := w1+v0; x

    end;

begin w1 := v0; Q1

end;

procedure P0;

    var v1;

    procedure P1;

        var v2;

    begin v2 := v1+v0; Q0(P0)

    end;

begin v1 := vo; P1

end;

begin {main program) v0 := 0; P0

end.
```


dynamic   static
link   link
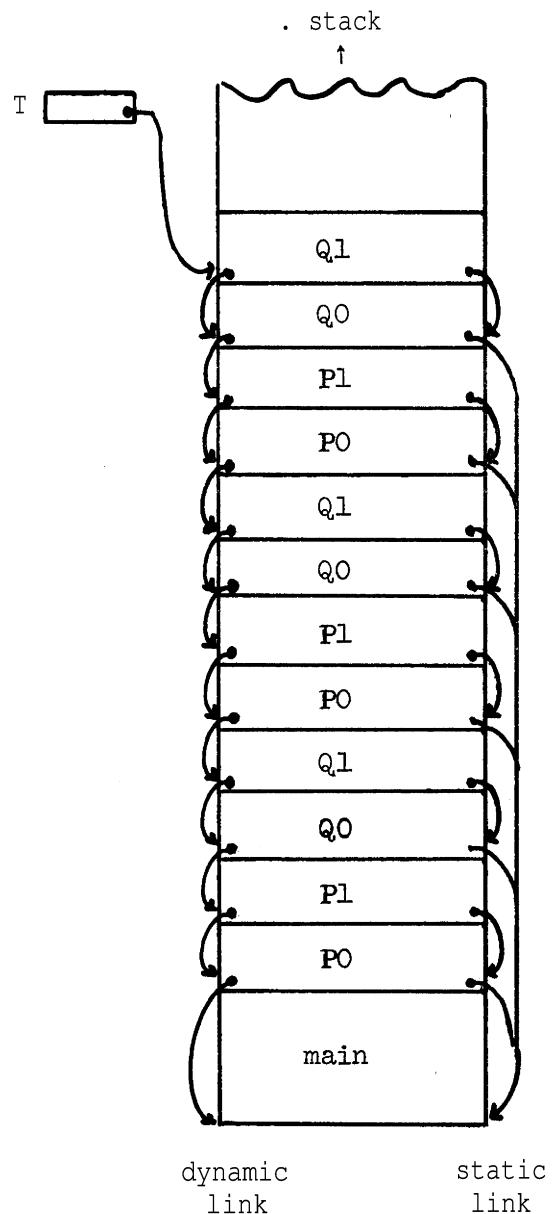
Method III:

Although the use of a high-speed index register to represent the origin of the link chains improves access speed significantly, the process of descending down the static chain to the record (data segment) with the desired level is relatively time-consuming. An ingenious device to reduce access time was introduced by Dijkstra [4] and is now widely used in compilers for block-structured languages. The device

25

is an array of base addresses, called the Display D , which is at any
time a copy of the static chain. If an object at level i is to be
accessed, the origin address of its data segment is quickly obtained
as D_i . The method is particularly attractive for computers with a
set of high-speed index registers which can be used as the Display.
The price for this increase in access speed -- apart from the reservation
of registers -- is the setting and updating of the Display each time a
procedure is called and terminated. To be more specific, the necessary
actions are as follows:

1. if an actual procedure of level i is called, D_i has to be
    set;

2. if control is returned from a procedure at level i to one
    at level j , $(j \geq i)$, $D_i \ldots D_j$ have to be reassigned;

3. if a formal procedure at level i is called from a procedure
    at level j , $D_i \ldots D_k$ have to be reassigned, where k is
    the level on which the static link emerging from the calling
    and the called procedures merge. Since k is not known at
    the time the procedure declaration is compiled, k can be
    chosen as zero without significant loss in efficiency.

This scheme was used in the implementation of PASCAL 6000. It is
described in Reference 4. Registers B1 ... B5 are used as the Display,
B5 is the origin of the link chains, and B6 is the pointer to the top
of the stack. The compiled instructions are the following:

26

Procedure call of P :

```
          SX7    L              save return address
          EQ     P              and jump
     ─────────────────────
     L  SBj    B5          ⎫
        SA1    B5          ⎪
        SB(j-1) X1  I       ⎬   update the display, if j ≥ i
        ........          ⎪
        SA1    X1          ⎪
        SBi    X1          ⎭
```

Procedure entry:

```
     P_F  SBi    X1         ⎫   prolog, entry for calls
          SA1    X1         ⎪   of formal procedures
          SB(i-1) X1        ⎬
          ........          ⎪
          SA1    X1         ⎪   update display
          SB1    X1         ⎭
```

```
     P_A  SA7    B5+2           save return address in header
          SX7    B(i-1)   ⎫
          SA7    B6       ⎬     save static link
          SX7    B5       ⎫
          SA7    B6+1     ⎬     save dynamic link
          SBi    B6             new display entry
          SB5    B6             T
          SB6    B6+L           top of stack, L = data segment length
```

Procedure exit:

```
          SB6    B5       ⎫     reset top of stack
          SA1    B5+1     ⎬
          SB5    X1       ⎭     reset T
          SA1    B5+2     ⎫
          SB7    X1       ⎬     fetch return address and jump
          JP     B7+0     ⎭
```

Notice that global variables in the main program are assigned absolute
addresses. Since B0 ≡ 0 , they can be considered as based on B0 .

In the first half of 1971, Prof. C. A. R. Hoare and his collaborators modified and bootstrapped the PASCAL compiler for the ICL 1966 computer [6]. One of the more significant alterations concerned the elimination of the Display, due to the fact that the ICL computer has no set of index registers that are available for a Display, and since the use of a Display was not considered to be an advantage, in this case. During a visit of Prof. Hoare in July 1971, he suggested that maybe even with a register set available for the Display, the benefits gained should be investigated. His suggestion was certainly valid, since variables either global or local to the most recently called procedure could be accessed with the same speed even without a Display. Thus the gain from a Display is limited to faster access of objects at intermediate levels, while the price is the updating at every call regardless of whether such objects are accessed or not. A superficial look at the PASCAL compiler itself showed that accesses to such intermediate level objects were indeed relatively rare, and it was decided to generate a version that would not use a Display (Method II). This version still uses the address register B5 as origin of the link chains (and base address of the most local data segment) and B6 as pointer to the top of the stack. The generated code is:

Procedure call P

```
    * X6 := base of environment of P
      SX7   L
      EQ    P
  L   ...
```

Procedure entry:

```
        SXO   B5    ⎫
        LXO   18    ⎪  pack and store
        BX7   X7+XO ⎬  dynamic link and
        SB5   B6    ⎪  return address
        SA7   B5+1  ⎭
        SB6   B7+L     stack pointer
    *   SA6   B5        static link
```

Procedure exit:

```
        SA1   B5+1  ⎫
        SB6   B5    ⎪  fetch and unpack
        SB7   X1    ⎬  dynamic link and
        LX1   42    ⎪  return address
        SB5   X1    ⎭
        JP    B7+0
```

Fetching an object x at level j from code at level i :

1)   j = 0:   SA1   BO+x

2)   j = i: SA1   B5+x

3)   O < j < i :   SA1   B 5
                   SA1   X1   repeated i-j-1 times
                   SA1   X1+x

A comparison of the codes generated by the two compilers shows that gains
and losses of execution speed should be measured, but also those of code
length. The shorter codes for procedure entry (2 - $2\frac{1}{4}$ words vs.
4 - 6 words), procedure exit (2 vs. 3 words), and procedure calls
(no updating of display) are very attractive, particularly in a compiler
where space is more on a premium than time. (It should be noted that
the instructions marked with an asterisk can be omitted in the call or
the entry code of procedures declared on the first level). Of course
it must be kept in mind that the decision about which compiler is to be
preferred depends not only on the weighting of space vs. time, but even
more on the programs to be processed. But it is obvious that if the

majority of these programs rarely use nested procedure declarations, and often call procedures on the same level, then the compiler <u>without</u> Display is to be preferred. The compiler itself, although featuring nested procedure declarations, but seldom accessing intermediate level variables, belongs to this class. Comparisons of code generated by the two compiler versions produced the following results:

1. The efficiency of codes not using a Display is in the average slightly higher (the compiler itself runs about 1.5% faster).

2. The size of codes not using a Display is smaller (by about 4% measured on 25 sample programs, about 6% in the case of the compiler's code).

3. The compiler program itself is slightly less complex without Display.

This episode where a more sophisticated method was abandoned in favor of a simpler and more direct technique could well be added to the list of D. Knuth's examples of adverse influences of "computer science" on "computer usage" [5]. Their common characteristic is that improved methods are adopted without closer inspection of the nature and direction of the improvement, and without analysis of the circumstances to be improved. An interesting fact is that the Burroughs B5500 computer -- specifically designed for ALGOL implementation -- did contain exactly the two base registers required to efficiently address objects at levels 0 and i . Unfortunately, addressing of intermediate level objects was impossible due to the software; this deficiency was justifiably criticized. The remedy adopted in the successor B6500 was, however, not a correction of the deficient software, but the inclusion of a full set of high-speed registers to serve as Display.

7.   Summary of the Main Trouble Spots of the CDC 6000 Architecture

1.   Use of one's complement arithmetic. In order to keep comparisons
      simple and efficient, the occurrence of negative zeroes must be
      prohibited.   (Note that PL and NG test the sign bit only.) Various
      optimizations are more cumbersome and less effective, because
      negative zeroes must be suppressed by additional instructions.
      Some instructions are themselves unsafe against the generation
      of -0 !

2.   No overflow check on fixed-point arithmetic. This lack is very
      serious and may cause wrong restuls in totally unexpected
      situations. Overflow check by software is prohibitive.

3.   No compare instructions. The use of subtraction may cause wrong
      results, unless expensive precautions are taken.

4.   Use of 48-bit multiplier and divider for fixed-point 60-bit numbers
      without warning of possible "overflow" of operands.

5.   Floating-point addition and subtraction without automatic post-
      normalization.

6.   Floating-point arithmetic with rounding of operands instead of
      rounding postnormalized results.

7.   No subroutine jump instruction depositing the program counter P
      in an operand register, and no return jump loading P from a general
      operand register. This defect requires the use of 3 instructions
      each to jump and deposit a return address, and to retrieve it and
      return, whereas many other computers need only a single instruction
      for these purposes.

Conclusions

When considering these complaints, the reader should bear in mind that this computer's architecture was conceived in the very early 1960's. The CDC 6600 machine was a very advanced design for a special purpose: fast number crunching. The design relied heavily on the use of several arithmetic units working simultaneously ("in parallel"). Integer arithmetic was considered as almost dispensible, and overflow interrupts as undesirable, because of the impossibility to mirror the present state of the entire machine by a simple program counter and of resuming computation. The use of simultaneously operating units is apparently also made responsible for the otherwise incomprehensible absence of post-normalization, namely because the unit for floaint-point addition does not contain a left-shift circuitry. A few years later, the CDC 6400 (and 6500) computers were announced; they were to have the same instruction set as the 6600, but only one conventional integrated arithmetic-logical unit. Although the "reasons" for the absence of interrupts and post-normalization had vanished, these "features" were retained in the name of compatibility. It was apparently considered most important that pitfall loaded programs could be transported to the new machines at no extra cost. This policy of staying "upward compatible with all previous mistakes" was sternly maintained when the successor to the 6000 series was announced in 1971.

This attitude, which is by no means atypical among computer manufacturers, makes it doubtful whether any progress toward more reliable and more efficient computing will ever be achievable. It does not seem so, until the computer consumers' attitudes will no longer justify the present manufacturers' policies. They, in turn, will not

change before they are made aware of ~~the hidden cost~~ involved in using the present equipment. I am convinced that the cost incurred by the programmers having to discover bugs the hard way by reprogramming repeatedly, and having to reexecute programs many times until they were believed to be correct, is incomparably higher than the reduction in cost due to staying compatible with outdated architectures. The project to develop the PASCAL compiler for the CDC 6000 computer unfortunately provided ample support for this conviction.

Acknowledgments

References

[1] N. Wirth, "The programming language PASCAL", ACTA INFORMATICA, Vol. 1, 35-68 (1971).

[2] D. S. Lindsay, "A rounded arithmetic FORTRAN compiler for CDC 6000 machines", U. of California, Berkeley, Dec. 1971.

[3] B. Randell and L. Russell, "ALGOL 60 implementation", Acad. Press, 1964.

[4] N. Wirth, "The design of a PASCAL compiler", Software - Practice and Experience, Vol. 1, ... (1971) .

[5] D. E. Knuth, "The dangers of computer-science theory", unpublished paper, August 1971.

[6] J. Welsh and C. Quinn, "A PASCAL compiler for ICL 1900 series computers", Dept. of Computer Science, Queen's University, Belfast, Sept. 1971.

```
005001  {$C+    T1: EXPRESSIONS  AND  ASSIGNMEWS }
005001     VAR I,J,K: INTEGER;
005004         X,Y,Z: REAL;
005007         N: 0..9999;
005010         P,Q: BOOLEAN;
005012   BEGIN { REAL ARITHMETIC 3
005076     x := 1.0; Y := x + 3.14159; Z := X*Y + X/Y;
005105     x := x + (Y + (Z + (1.0 + X)));
005112     x a= ABS(+Y);  Y := SQR(X); Z := -X;
005117     {$R+ ROUNDED REAL ARITHMETIC}
005117     X := Y + Z; X := Y*Z; X := Y/Z;
005127     { INTEGER ARITHMETIC }
005127     I := 1; J := I + 100; K := I * J; K := I DIV J;
005140     K := (-J) MOD K;  J a= SQR(J);
005145     I := TRUNC(X) ; Z := I;  X := I/J;
005154     { BOOLEAN ARITHMETIC 3
005154     P := TRUE; Q:= P ∧ ¬(Q∨P);
005160     P := x = Y; P := I = J; Q := P = Q;
005171     P := X < Y; P := I < J; Q := P < Q;
005200     P a= x ≤ Y; P:= I ≤ J; Q := P ≤ Q;
005210     Q := ODD(I);
005212     { OPTIMIZATION OF INTEGER ARITHMETIC }
005212     I a= I*8 + J*10;
005216     J := I DIV 8 - N OIV 2; K := I MOD 16;
005223     N := I + 100
005223   END  •
```

```
005074                                005103
                                          SA3    B0+005005
    SA7    B5+B0                          FX2    X2/X3
    SX7    B5+B0                          FX1    X1+X2
    SA7    B0+005000
005075                                005104
                                          NX6    B0,X1
    SB6    B5+000001                      SA6    B0+005006
    SA1    B0+005225                      NO
005076                                005105
    BX6    Xi                            SA1    B0+005004
    SA6    B0+005004                      SA2    B0+005005
    NO
005077                                005106
                                          SA3    B0+005006
    SA1    B0+005004                      SA4    B0+005225
    SA2    B0+005226
005100                                005107
                                          SA5    B0+005004
    FX1    X1+X2                          FX4    x4+x5
    NX6    B0,X1                          NX4    B0,X4
    SA6    B0+005005
005101                                005110
                                          FX3    x 3 + x 4
    SAI    B0+005004                      NX3    B0,X3
    SA2    B0+005005                      FX2    X2+X3
005102                                    NX2    B0,X2
    FX1    X1*X2                       005111
    SA2    B0+005004
    NO                                    FX1    X1+X2
                                          NX6    B0,X1
                                          SA6    B0+005004
```

005112		
	SAI	B0+005005
	BXO	xi
	AXO	73
005113		
	BX6	XO-X1
	SA6	B0+005004
	NO	
005114		
	SA1	B0+005004
	FX6	X1*X1
	NO	
005115		
	SA6	B0+005005
	SA1	B0+005004
005116		
	BXO	XO-XO
	I X6	XO-X1
	SA6	B0+005006
005117		
	SA1	B0+005005
	SA2	B0+005006
005120		
	FXO	X1+X2
	NXO	B0,X0
	DX1	X1+X2
	RX1	XO+X 1
005121		
	NX6	B0,X1
	SA6	B0+005004
	NO	
005122		
	SA1	B0+005005
	SA2	B0+005006
005123		
	FXO	X1*X2
	DX1	X1*X2
	RX6	XO+X1
	NO	
005124		
	SA6	B0+005004
	SA1	B0+005005
005125		
	SA2	B0+005006
	RX6	X1/X2
	NO	
005126		
	SA6	B0+005004
	SX6	B0+000001
005127		
	SA6	B0+005001

005130	SA1	B0+005001 --
	SX0	B0+000144
	I X6	X1+X0
	NO	
005131		
	SA6	B0+005002
	SA1	B0+005001
005132		
	SA2	BP+005002
	DX1	X1*X2
	0x0	XO-XO
005133		
	IX6	X1+X0
	SA6	B0+005003
	NO	
005134		
	SA1	B0+005001
	SA2	B0+005002
005135		
	PX2	B0,X2
	NX2	B0,X2
	PX1	B0,X1
	FXI	X1/X2
005136		
	UX1	B7,X1
	LX1	B7,X1
	0x0	XO-XO
	I X6	X1+X 0
005137		
	SA6	B0+005003
	SAI	B0+005002
005140		
	BXO	XO-XO
	IX1	XO-X 1
	SA2	B0+005003
005141		
	PX6	B0,X2
	NX6	B0,X6
	PXO	B0,X1
	FX6	XO/X 6
005142		
	UX6	B7,X6
	LX6	87, X6
	DX6	X2*X6
	I X6	X1-X6
065143		
	SA6	B0+005003
	SA1	B0+005002
005144		
	DX6	X1*X1
	SA6	B0+005002
	NO	

```
005145  ─────────────
        SA1   B0+005004
        UX1   B7,X1
        1x1   B7,X1
005146
        0x0   X0-X0
        I X6  X1+X0
        S A6  B0+005001
005147  ─────────────
        SA1   B0+005001
        BX6   X1
        P X6  B0,X6
005150
        NX6   B0,X6
        S A6  B0+005006
        NO
005151  ─────────────
        SA1   B0+005001
        SA2   B0+005002
005152
        PX2   B0,X2
        NX2   B0,X2
        PX1   B0,X1
        NX1   B0,X1
005153
        RX6   X1/X2
        S A6  B0+005004
        NO
005154  ─────────────
        SX6   B0+000001
        S A6  B0+005010
005155  ─────────────
        SA1   B0+005010
        SA2   B0+005011
005156
        S A3  B0+005010
        0x2   X2vX3
        MX0   73
005157
        0x2   ¬X2-X0
        BX6   X1^X2
        S A6  B0+005011
005160  ─────────────
        SA1   B0+005004
        SA2   B0+005005
005161
        IX0   X1-X2
        MX6   00
        NZ    X0,005163
005162
        S X6  B0+000001
        NO
```

```
        NO
005163
        SA6   B0+005010  ─────────────
        SA1   B0+005001
005164
        SA2   B0+00500 2
        1x0   X1-X2
        MX6   00
005165
        NZ    X0,005166
        S X6  B0+000001
005166
        S A6  B0+005010  ─────────────
        SA1   B0+005010
005167
        SA2   B0+005011
        BX1   X1-X2
        MX0   73
005170
        BX6   ¬X1-X0
        S A6  B0+005011
        NO
005171  ─────────────
        SA1   B0+005004
        SA2   B0+005005
005172
        FX1   X1-X2
        MX0   01
        %X6   X0^X1
        LX6   01
005173
        SA6   B0+005010  ─────────────
        SA1   B0+00500 1
005174
        SA2   B0+005002
        1x1   X1-X2
        HX0   01
005175
        BX6   X0^X1
        LX6   01
        SA6   B0+005010
005176  ─────────────
        SA1   B0+005010
        SA2   B0+0050 11
005177
        BX6   ¬X1^X2
        S A6  B0+005011
        NO
```

37

005200		
	SA1	B0+005004
	SA2	B0+005005
005201		
	FX1	X2-X1
	HX0	01
	BX6	¬X1∧X0
	LX6	01
005202		
	SA6	B0+005010
	SA1	004005001
005203		
	SA2	B0+005002
	1X1	X2-X1
	MX0	01
005204		
	BX6	¬X1∧X0
	LX6	01
	SA6	B0+005010
005205		
	SA1	B0+005010
	SA2	B0+005011
005206		
	BX1	¬X2∧X1
	MX0	73
	BX6	¬X1-X0
	NO	
005207		
	SA6	B0+005011
	SA1	B0+005001
005210		
	BX0	X1
	LX0	73
	BX1	X1-X0
	MX0	01
005211		
	BX6	X0∧X1
	LX6	01
	SA6	B0+005011
005212		
	SA1	B0+005001
	LX1	03
	NO	
005213		
	SA2	B0+005002
	LX2	01
	BX0	X2
005214		
	LX2	02
	IX2	X2+X0
	IX6	X1+X2
	NO	
005215		
	SA6	B0+005001

	SA1	B0+005001
005216		
	AX1	03
	BX0	X0-X0
	IX1	X1+X0
	NO	
005217		
	SA2	804005007
	AX2	01
	IX6	X1-X2
005220		
	SA6	B0+005002
	SA1	B0+005001
005221		
	BX0	Xi
	AX0	04
	LX0	04
	IX6	X1-X0
005222		
	SA6	B0+005003
	SA1	B0+005001
005223		
	SX6	x14000144
	SA6	B0+005007
005224		
	SA1	B5+B0
	SB7	X1+B0
	JP	B7+000000
005225	17204000000000000000	
005226	17216220771740156064	