# Berkeley Pascal PX Implementation Notes
# Version 2.0 – January, 1979

*William N. Joy*†

*M. Kirk McKusick*‡

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

### Introduction

These *PX Implementation Notes* have been updated from the original PDP 11/70 implementation notes to reflect the interpreter that runs on the VAX 11/780. These notes consist of four major parts. The first part outlines the general organization of *px*. Section 2 describes the operations (instructions) of the interpreter while section 3 focuses on input/output related activity. A final section gives conclusions about the viability of an interpreter based approach to language implementation for instruction.

### Related Berkeley Pascal documents

The *PXP Implementation Notes* give details of the internals of the execution profiler *pxp;* parts of the interpreter related to *pxp* are discussed in section 2.10. A paper describing the syntactic error recovery mechanism used in *pi* was presented at the ACM Conference on Compiler Construction in Boulder Colorado in August, 1979.

### Acknowledgements

This version of *px* is a PDP 11/70 to VAX 11/780 opcode mapping of the original *px* that was designed and implemented by Ken Thompson, with extensive modifications and additions by William Joy and Charles Haley. Without their work, this Berkeley Pascal system would never have existed. These notes were first written by William Joy for the PDP 11/70 implementation. We would also like to thank our faculty advisor Susan L. Graham for her encouragement, her helpful comments and suggestions relating to Berkeley Pascal and her excellent editorial assistance.

† The financial support of the National Science Foundation under grants MCS74-07644-A03 and MCS78-07291 and of an IBM Graduate Fellowship are gratefully acknowledged.

‡ The financial support of a Howard Hughes Graduate Fellowship is gratefully acknowledged.

## 1. Organization

Most of *px* is written in the VAX 11/780 assembly language, using the UNIX® assembler *as.* Portions of *px* are also written in the UNIX systems programming language C. *Px* consists of a main procedure that reads in the interpreter code, a main interpreter loop that transfers successively to various code segments implementing the abstract machine operations, built-in procedures and functions, and several routines that support the implementation of the Pascal input-output environment.

The interpreter runs at a fraction of the speed of equivalent compiled C code, with this fraction varying from 1/5 to 1/15. The interpreter occupies 18.5K bytes of instruction space, shared among all processes executing Pascal, and has 4.6K bytes of data space (constants, error messages, etc.) a copy of which is allocated to each executing process.

### 1.1. Format of the object file

*Px* normally interprets the code left in an object file by a run of the Pascal translator *pi.* The file where the translator puts the object originally, and the most commonly interpreted file, is called *obj.* In order that all persons using *px* share a common text image, this executable file is a small process that coordinates with the interpreter to start execution. The interpreter code is placed at the end of a special ''header'' file and the size of the initialized data area of this header file is expanded to include this code, so that during execution it is located at an easily determined address in its data space. When executed, the object process creates a *pipe*, creates another process by doing a *fork*, and arranges that the resulting parent process becomes an instance of *px*. The child process then writes the interpreter code through the pipe that it has to the interpreter process parent. When this process is complete, the child exits.

The real advantage of this approach is that it does not require modifications to the shell, and that the resultant objects are ''true objects'' not requiring special treatment. A simpler mechanism would be to determine the name of the file that was executed and pass this to the interpreter. However it is not possible to determine this name in all cases.‡

### 1.2. General features of object code

Pascal object code is relocatable as all addressing references for control transfers within the code are relative. The code consists of instructions interspersed with inline data. All instructions have a length that is an even number of bytes. No variables are kept in the object code area.

The first byte of a Pascal interpreter instruction contains an operation code. This allows a total of 256 major operation codes, and 232 of these are in use in the current *px.* The second byte of each interpreter instruction is called the ''sub-operation code'', or more commonly the *sub-opcode.* It contains a small integer that may, for example, be used as a block-structure level for the associated operation. If the instruction can take a longword constant, this constant is often packed into the sub-opcode if it fits into 8 bits and is not zero. A sub-opcode value of zero specifies that the constant would not fit and therefore follows in the next word. This is a space optimization, the value of zero for flagging the longer case being convenient because it is easy to test.

Other instruction formats are used. The branching instructions take an offset in the following word, operators that load constants onto the stack take arbitrarily long inline constant values, and many operations deal exclusively with data on the interpreter stack, requiring no inline data.

### 1.3. Stack structure of the interpreter

The interpreter emulates a stack-structured Pascal machine. The ''load'' instructions put values onto the stack, where all arithmetic operations take place. The ''store'' instructions take values off the stack and place them in an address that is also contained on the stack. The only way to move data or to compute in

‡ For instance, if the *pxref* program is placed in the directory '/usr/bin' then when the user types ''pxref program.p'' the first argument to the program, nominally the programs name, is ''pxref.'' While it would be possible to search in the standard place, i.e. the current directory, and the system directories '/bin' and '/usr/bin' for a corresponding object file, this would be expensive and not guaranteed to succeed. Several shells exist that allow other directories to be searched for commands, and there is, in general, no way to determine what these directories are.

the machine is with the stack.

To make the interpreter operations more powerful and to thereby increase the interpreter speed, the arithmetic operations in the interpreter are "typed". That is, length conversion of arithmetic values occurs when they are used in an operation. This eliminates interpreter cycles for length conversion and the associated overhead. For example, when adding an integer that fits in one byte to one that requires four bytes to store, no "conversion" operators are required. The one byte integer is loaded onto the stack, followed by the four byte integer, and then an adding operator is used that has, implicit in its definition, the sizes of the arguments.

### 1.4. Data types in the interpreter

The interpreter deals with several different fundamental data types. In the memory of the machine, 1, 2, and 4 byte integers are supported, with only 2 and 4 byte integers being present on the stack. The interpreter always converts to 4 byte integers when there is a possibility of overflowing the shorter formats. This corresponds to the Pascal language definition of overflow in arithmetic operations that requires that the result be correct if all partial values lie within the bounds of the base integer type: 4 byte integer values.

Character constants are treated similarly to 1 byte integers for most purposes, as are Boolean values. All enumerated types are treated as integer values of an appropriate length, usually 1 byte. The interpreter also has real numbers, occupying 8 bytes of storage, and sets and strings of varying length. The appropriate operations are included for each data type, such as set union and intersection and an operation to write a string.

No special **packed** data formats are supported by the interpreter. The smallest unit of storage occupied by any variable is one byte. The built-ins *pack* and *unpack* thus degenerate to simple memory to memory transfers with no special processing.

### 1.5. Runtime environment

The interpreter runtime environment uses a stack data area and a heap data area, that are kept at opposite ends of memory and grow towards each other. All global variables and variables local to procedures and functions are kept in the stack area. Dynamically allocated variables and buffers for input/output are allocated in the heap.

The addressing of block structured variables is done by using a fixed display that contains the address of its stack frame for each statically active block.† This display is referenced by instructions that load and store variables and maintained by the operations for block entry and exit, and for non-local **goto** statements.

### 1.6. Dp, lc, loop

Three "global" variables in the interpreter, in addition to the "display", are the *dp, lc,* and the *loop.* The *dp* is a pointer to the display entry for the current block; the *lc* is the abstract machine location counter; and the *loop* is a register that holds the address of the main interpreter loop so that returning to the loop to fetch the next instruction is a fast operation.

### 1.7. The stack frame structure

Each active block has a stack frame consisting of three parts: a block mark, local variables, and temporary storage for partially evaluated expressions. The stack in the interpreter grows from the high addresses in memory to the low addresses, so that those parts of the stack frame that are "on the top" of the stack have the most negative offsets from the display entry for the block. The major parts of the stack frame are represented in Figure 1.1. Note that the local variables of each block have negative offsets from the corresponding display entry, the "first" local variable having offset '−2'.

### 1.8. The block mark

The block mark contains the saved information necessary to restore the environment when the current block exits. It consists of two parts. The first and top-most part is saved by the CALL instruction in the

† Here "block" is being used to mean any *procedure*, *function* or the main program.

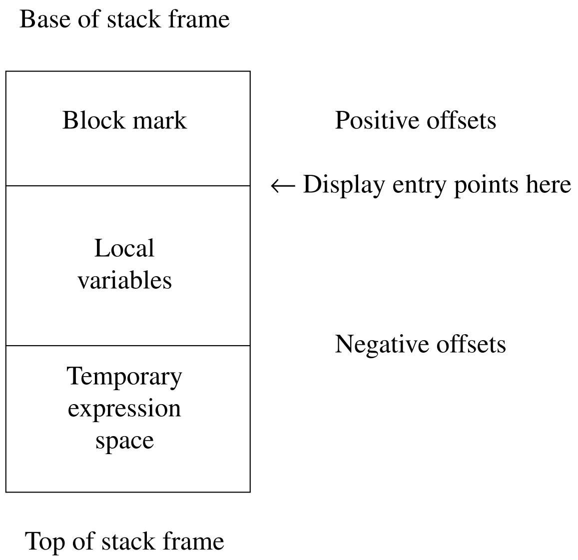


Figure 1.1 – Structure of stack frame

interpreter. This information is not present for the main program as it is never “called”. The second part of the block mark is created by the BEG begin block operator that also allocates and clears the local variable storage. The format of these blocks is represented in Figure 1.2.

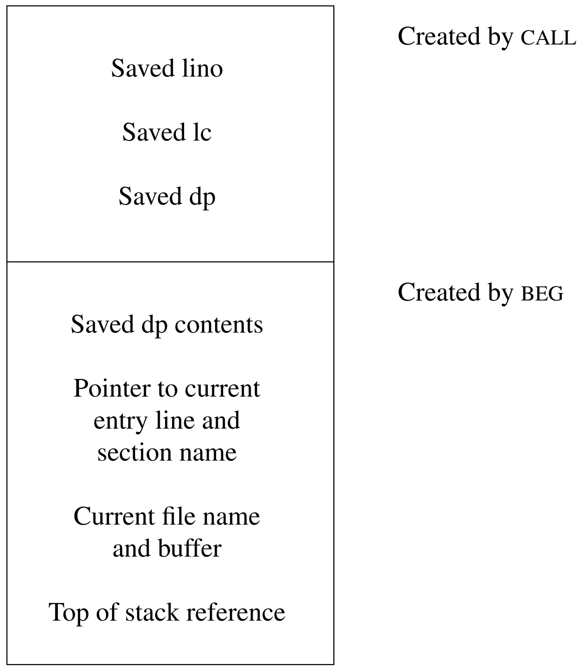


Figure 1.2 – Block mark structure

The data saved by the CALL operator includes the line number *lino* of the point of call, that is printed if the program execution ends abnormally; the location counter *lc* giving the return address; and the current display entry address *dp* at the time of call.

The BEG begin operator saves the previous display contents at the level of this block, so that the display can be restored on block exit. A pointer to the beginning line number and the name of this block is also saved. This information is stored in the interpreter object code in-line after the BEG operator. It is used in printing a post-mortem backtrace. The saved file name and buffer reference are necessary because of the input/output structure (this is discussed in detail in sections 3.3 and 3.4). The top of stack reference gives the value the stack pointer should have when there are no expression temporaries on the stack. It is used for a consistency check in the LINO line number operators in the interpreter, that occurs before each statement executed. This helps to catch bugs in the interpreter, that often manifest themselves by leaving the stack non-empty between statements.

Note that there is no explicit static link here. Thus to set up the display correctly after a non-local **goto** statement one must ‘‘unwind’’ through all the block marks on the stack to rebuild the display.

### 1.9. Arguments and return values

A function returns its value into a space reserved by the calling block. Arguments to a **function** are placed on top of this return area. For both **procedure** and **function** calls, arguments are placed at the end of the expression evaluation area of the caller. When a **function** completes, expression evaluation can continue after popping the arguments to the **function** off the stack, exactly as if the function value had been ‘‘loaded’’. The arguments to a **procedure** are also popped off the stack by the caller after its execution ends.

As a simple example consider the following stack structure for a call to a function *f,* of the form ‘‘f(a)’’.

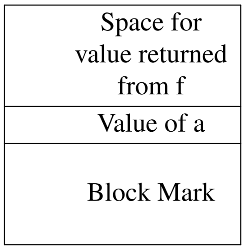


Figure 1.3 − Stack structure on function call ‘f(a)’

If we suppose that *f* returns a *real* and that *a* is an integer, the calling sequence for this function would be:

```
PUSH     −8
RV4:l    a
CALL:l   f
POP      4
```

Here we use the operator PUSH to clear space for the return value, load *a* on the stack with a ‘‘right value’’ operator, call the function, pop off the argument *a*, and can then complete evaluation of the containing expression. The operations used here will be explained in section 2.

If the function *f* were given by

```
function f(i: integer): real;
begin
    f := i
end;
```

then *f* would have code sequence:

```
BEG:2   0
        11
        "f"
LV:l    40
RV4:l   32
AS48
END
```

Here the BEG operator takes 9 bytes of inline data. The first byte specifies the length of the function name. The second longword specifies the amount of local variable storage, here none. The succeeding two lines give the line number of the **begin** and the name of the block for error traceback. The BEG operator places a name pointer in the block mark. The body of the **function** first takes an address of the **function** result variable *f* using the address of operator LV *a*. The next operation in the interpretation of this function is the loading of the value of *i*. *I* is at the level of the **function** *f*, here symbolically *l,* and the first variable in the local variable area. The **function** completes by assigning the 4 byte integer on the stack to the 8 byte return location, hence the AS48 assignment operator, and then uses the END operator to exit the current block.

### 1.10. The main interpreter loop

The main interpreter loop is simply:

```
iloop:
        caseb   (lc)+,$0,$255
        <table of opcode interpreter addresses>
```

The main opcode is extracted from the first byte of the instruction and used to index into the table of opcode interpreter addresses. Control is then transferred to the specified location. The sub-opcode may be used to index the display, as a small constant, or to specify one of several relational operators. In the cases where a constant is needed, but it is not small enough to fit in the byte sub-operator, a zero is placed there and the constant follows in the next word. Zero is easily tested for, as the instruction that fetches the sub-opcode sets the condition code flags. A construction like:

```
_OPER:
        cvtbl   (lc)+,r0
        bneq    L1
        cvtwl   (lc)+,r0
L1:     ...
```

is all that is needed to effect this packing of data. This technique saves space in the Pascal *obj* object code.

The address of the instruction at *iloop* is always contained in the register variable *loop*. Thus a return to the main interpreter is simply:

```
        jmp   (loop)
```

that is both quick and occupies little space.

### 1.11. Errors

Errors during interpretation fall into three classes:

1) Interpreter detected errors.
2) Hardware detected errors.
3) External events.

Interpreter detected errors include I/O errors and built-in function errors. These errors cause a subroutine call to an error routine with a single parameter indicating the cause of the error. Hardware errors such as range errors and overflows are fielded by a special routine that determines the opcode that caused the error. It then calls the error routine with an appropriate error parameter. External events include interrupts and system limits such as available memory. They generate a call to the error routine with an appropriate error code. The error routine processes the error condition, printing an appropriate error message and usually a backtrace from the point of the error.

## 2. Operations

### 2.1. Naming conventions and operation summary

Table 2.1 outlines the opcode typing convention. The expression "a above b" means that 'a' is on top of the stack with 'b' below it. Table 2.3 describes each of the opcodes. The character '∗' at the end of a name specifies that all operations with the root prefix before the '∗' are summarized by one entry. Table 2.2 gives the codes used to describe the type inline data expected by each instruction.

Table 2.1 − Operator Suffixes

Unary operator suffixes

| Suffix | Example | Argument type |
|---|---|---|
| 2 | NEG2 | Short integer (2 bytes) |
| 4 | SQR4 | Long integer (4 bytes) |
| 8 | ABS8 | Real (8 bytes) |

Binary operator suffixes

| Suffix | Example | Argument type |
|---|---|---|
| 2 | ADD2 | Two short integers |
| 24 | MUL24 | Short above long integer |
| 42 | REL42 | Long above short integer |
| 4 | DIV4 | Two long integers |
| 28 | DVD28 | Short integer above real |
| 48 | REL48 | Long integer above real |
| 82 | SUB82 | Real above short integer |
| 84 | MUL84 | Real above long integer |
| 8 | ADD8 | Two reals |

Other Suffixes

| Suffix | Example | Argument types |
|---|---|---|
| T | ADDT | Sets |
| G | RELG | Strings |

Table 2.2 – Inline data type codes

| Code | Description |
|---|---|
| *a* | An address offset is given in the word following the instruction. |
| *A* | An address offset is given in the four bytes following the instruction. |
| *l* | An index into the display is given in the sub-opcode. |
| *r* | A relational operator is encoded in the sub-opcode. (see section 2.3) |
| *s* | A small integer is placed in the sub-opcode, or in the next word if it is zero or too large. |
| *v* | Variable length inline data. |
| *w* | A word value in the following word. |
| *W* | A long value in the following four bytes. |
| " | An inline constant string. |

| Table 2.3 – Machine operations | | |
|---|---|---|
| Mnemonic | Reference | Description |
| ABS* | 2.7 | Absolute value |
| ADD* | 2.7 | Addition |
| AND | 2.4 | Boolean and |
| ARGC | 2.14 | Returns number of arguments to current process |
| ARGV | 2.14 | Copy specified process argument into char array |
| AS* | 2.5 | Assignment operators |
| ASRT | 2.12 | Assert *true* to continue |
| ATAN | 2.13 | Returns arctangent of argument |
| BEG s,W,w," | 2.2,1.8 | Write second part of block mark, enter block |
| BUFF | 3.11 | Specify buffering for file "output" |
| CALL l,A | 2.2,1.8 | Procedure or function call |
| CARD s | 2.11 | Cardinality of set |
| CASEOP* | 2.9 | Case statements |
| CHR* | 2.15 | Returns integer to ascii mapping of argument |
| CLCK | 2.14 | Returns user time of program |
| CON* v | 2.5 | Load constant operators |
| COS | 2.13 | Returns cos of argument |
| COUNT w | 2.10 | Count a statement count point |
| CTTOT s,w,w | 2.11 | Construct set |
| DATE | 2.14 | Copy date into char array |
| DEFNAME | 3.11 | Attach file name for **program** statement files |
| DISPOSE | 2.15 | Dispose of a heap allocation |
| DIV* | 2.7 | Fixed division |
| DVD* | 2.7 | Floating division |
| END | 2.2,1.8 | End block execution |
| EOF | 3.10 | Returns *true* if end of file |
| EOLN | 3.10 | Returns *true* if end of line on input text file |
| EXP | 2.13 | Returns exponential of argument |
| EXPO | 2.13 | Returns machine representation of real exponent |
| FILE | 3.9 | Push descriptor for active file |
| FLUSH | 3.11 | Flush a file |
| FNIL | 3.7 | Check file initialized, not eof, synced |
| FOR* a | 2.12 | For statements |
| GET | 3.7 | Get next record from a file |
| GOTO l,A | 2.2,1.8 | Non-local goto statement |
| HALT | 2.2 | Produce control flow backtrace |
| IF a | 2.3 | Conditional transfer |
| IN s,w,w | 2.11 | Set membership |
| INCT | 2.11 | Membership in a constructed set |
| IND* | 2.6 | Indirection operators |
| INX* s,w,w | 2.6 | Subscripting (indexing) operator |
| ITOD | 2.12 | Convert integer to real |
| ITOS | 2.12 | Convert integer to short integer |
| LINO s | 2.2 | Set line number, count statements |
| LLIMIT | 2.14 | Set linelimit for output text file |
| LLV l,W | 2.6 | Address of operator |
| LN | 2.13 | Returns natural log of argument |
| LRV* l,A | 2.5 | Right value (load) operators |
| LV l,w | 2.6 | Address of operator |
| MAX s,w | 3.8 | Maximum of top of stack and *w* |
| MESSAGE | 3.6 | Write to terminal |

| Table 2.3 – Machine operations | | |
|---|---|---|
| Mnemonic | Reference | Description |
| MIN s | 3.8 | Minimum of top of stack and *s* |
| MOD* | 2.7 | Modulus |
| MUL* | 2.7 | Multiplication |
| NAM A | 3.8 | Convert enumerated type value to print format |
| NEG* | 2.7 | Negation |
| NEW s | 2.15 | Allocate a record on heap, set pointer to it |
| NIL | 2.6 | Assert non-nil pointer |
| NODUMP s,W,w," | 2.2 | BEG main program, suppress dump |
| NOT | 2.4 | Boolean not |
| ODD* | 2.15 | Returns *true* if argument is odd, *false* if even |
| OFF s | 2.5 | Offset address, typically used for field reference |
| OR | 2.4 | Boolean or |
| PACK s,w,w,w | 2.15 | Convert and copy from unpacked to packed |
| PAGE | 3.8 | Output a formfeed to a text file |
| POP s | 2.2,1.9 | Pop (arguments) off stack |
| PRED* | 2.7 | Returns predecessor of argument |
| PUSH s | 2.2,1.9 | Clear space (for function result) |
| PUT | 3.8 | Output a record to a file |
| PXPBUF w | 2.10 | Initialize *pxp* count buffer |
| RANDOM | 2.13 | Returns random number |
| RANG* v | 2.8 | Subrange checking |
| READ* | 3.7 | Read a record from a file |
| REL* r | 2.3 | Relational test yielding Boolean result |
| REMOVE | 3.11 | Remove a file |
| RESET | 3.11 | Open file for input |
| REWRITE | 3.11 | Open file for output |
| ROUND | 2.13 | Returns TRUNC(argument + 0.5) |
| RV* l,a | 2.5 | Right value (load) operators |
| SCLCK | 2.14 | Returns system time of program |
| SDUP | 2.2 | Duplicate top stack word |
| SEED | 2.13 | Set random seed, return old seed |
| SIN | 2.13 | Returns sin of argument |
| SQR* | 2.7 | Squaring |
| SQRT | 2.13 | Returns square root of argument |
| STLIM | 2.14 | Set program statement limit |
| STOD | 2.12 | Convert short integer to real |
| STOI | 2.12 | Convert short to long integer |
| SUB* | 2.7 | Subtraction |
| SUCC* | 2.7 | Returns successor of argument |
| TIME | 2.14 | Copy time into char array |
| TRA a | 2.2 | Short control transfer (local branching) |
| TRA4 A | 2.2 | Long control transfer |
| TRACNT w,A | 2.10 | Count a procedure entry |
| TRUNC | 2.13 | Returns integer part of argument |
| UNDEF | 2.15 | Returns *false* |
| UNIT* | 3.10 | Set active file |
| UNPACK s,w,w,w | 2.15 | Convert and copy from packed to unpacked |
| WCLCK | 2.14 | Returns current time stamp |
| WRITEC | 3.8 | Character unformatted write |
| WRITEF l | 3.8 | General formatted write |
| WRITES l | 3.8 | String unformatted write |

| Table 2.3 – Machine operations | | |
|---|---|---|
| Mnemonic | Reference | Description |
| WRITLN | 3.8 | Output a newline to a text file |

## 2.2. Basic control operations

**HALT**

Corresponds to the Pascal procedure *halt*; causes execution to end with a post-mortem backtrace as if a run-time error had occurred.

**BEG s,W,w,"**

Causes the second part of the block mark to be created, and *W* bytes of local variable space to be allocated and cleared to zero. Stack overflow is detected here. *w* is the first line of the body of this section for error traceback, and the inline string (length s) the character representation of its name.

**NODUMP s,W,w,"**

Equivalent to BEG, and used to begin the main program when the ‘‘p’’ option is disabled so that the post-mortem backtrace will be inhibited.

**END**

Complementary to the operators CALL and BEG, exits the current block, calling the procedure *pclose* to flush buffers for and release any local files. Restores the environment of the caller from the block mark. If this is the end for the main program, all files are *flushed,* and the interpreter is exited.

**CALL l,A**

Saves the current line number, return address, and active display entry pointer *dp* in the first part of the block mark, then transfers to the entry point given by the relative address *A*, that is the beginning of a **procedure** or **function** at level *l.*

**PUSH s**

Clears *s* bytes on the stack. Used to make space for the return value of a **function** just before calling it.

**POP s**

Pop *s* bytes off the stack. Used after a **function** or **procedure** returns to remove the arguments from the stack.

**TRA a**

Transfer control to relative address *a* as a local **goto** or part of a structured statement.

**TRA4 A**

Transfer control to an absolute address as part of a non-local **goto** or to branch over procedure bodies.

**LINO s**

Set current line number to *s.* For consistency, check that the expression stack is empty as it should be (as this is the start of a statement.) This consistency check will fail only if there is a bug in the interpreter or the interpreter code has somehow been damaged. Increment the statement count and if it exceeds the statement limit, generate a fault.

**GOTO l,A**

Transfer control to address *A* that is in the block at level *l* of the display. This is a non-local **goto.** Causes each block to be exited as if with END, flushing and freeing files with *pclose,* until the current display entry is at level *l.*

**SDUP***

Duplicate the word or long on the top of the stack. This is used mostly for constructing sets. See section 2.11.

### 2.3. If and relational operators

**IF a**

The interpreter conditional transfers all take place using this operator that examines the Boolean value on the top of the stack. If the value is *true*, the next code is executed, otherwise control transfers to the specified address.

**REL* r**

These take two arguments on the stack, and the sub-operation code specifies the relational operation to be done, coded as follows with ‘a’ above ‘b’ on the stack:

| Code | Operation |
|---|---|
| 0 | a = b |
| 2 | a <> b |
| 4 | a < b |
| 6 | a > b |
| 8 | a <= b |
| 10 | a >= b |

Each operation does a test to set the condition code appropriately and then does an indexed branch based on the sub-operation code to a test of the condition here specified, pushing a Boolean value on the stack.

Consider the statement fragment:

**if** a = b **then**

If *a* and *b* are integers this generates the following code:

```
RV4:l   a
RV4:l   b
REL4    =
IF      Else part offset

  ... Then part code ...
```

### 2.4. Boolean operators

The Boolean operators AND, OR, and NOT manipulate values on the top of the stack. All Boolean values are kept in single bytes in memory, or in single words on the stack. Zero represents a Boolean *false*, and one a Boolean *true*.

### 2.5. Right value, constant, and assignment operators

**LRV* l,A**
**RV* l,a**

The right value operators load values on the stack. They take a block number as a sub-opcode and load the appropriate number of bytes from that block at the offset specified in the following word onto the stack. As an example, consider LRV4:

```
_LRV4:
        cvtbl   (lc)+,r0                    #r0 has display index
        addl3   _display(r0),(lc)+,r1       #r1 has variable address
        pushl   (r1)                        #put value on the stack
        jmp     (loop)
```

Here the interpreter places the display level in r0. It then adds the appropriate display value to the inline offset and pushes the value at this location onto the stack. Control then returns to the main interpreter loop. The RV* operators have short inline data that reduces the space required to address the first 32K of stack space in each stack frame. The operators RV14 and RV24 provide explicit conversion to long as the data is pushed. This saves the generation of STOI to align arguments to C subroutines.

**CON* r**

The constant operators load a value onto the stack from inline code. Small integer values are condensed and loaded by the CON1 operator, that is given by

```
_CON1:
        cvtbw   (lc)+,−(sp)
        jmp     (loop)
```

Here note that little work was required as the required constant was available at (lc)+. For longer constants, *lc* must be incremented before moving the constant. The operator CON takes a length specification in the sub-opcode and can be used to load strings and other variable length data onto the stack. The operators CON14 and CON24 provide explicit conversion to long as the constant is pushed.

**AS***

The assignment operators are similar to arithmetic and relational operators in that they take two operands, both in the stack, but the lengths given for them specify first the length of the value on the stack and then the length of the target in memory. The target address in memory is under the value to be stored. Thus the statement

i := 1

where *i* is a full-length, 4 byte, integer, will generate the code sequence

```
LV:l      i
CON1:1
AS24
```

Here LV will load the address of *i,* that is really given as a block number in the sub-opcode and an offset in the following word, onto the stack, occupying a single word. CON1, that is a single word instruction, then loads the constant 1, that is in its sub-opcode, onto the stack. Since there are not one byte constants on the stack, this becomes a 2 byte, single word integer. The interpreter then assigns a length 2 integer to a length 4 integer using AS24. The code sequence for AS24 is given by:

```
_AS24:
        incl    lc
        cvtwl   (sp)+,*(sp)+
        jmp     (loop)
```

Thus the interpreter gets the single word off the stack, extends it to be a 4 byte integer gets the target address off the stack, and finally stores the value in the target. This is a typical use of the constant and assignment operators.

### 2.6. Addressing operations

**LLV l,W**
**LV l,w**

The most common operation done by the interpreter is the "left value" or "address of" operation. It is given by:

```
_LLV:
        cvtbl   (lc)+,r0                    #r0 has display index
        addl3   _display(r0),(lc)+,−(sp)    #push address onto the stack
        jmp     (loop)
```

It calculates an address in the block specified in the sub-opcode by adding the associated display entry to the offset that appears in the following word. The LV operator has a short inline data that reduces the space required to address the first 32K of stack space in each call frame.

**OFF s**

The offset operator is used in field names. Thus to get the address of

p↑.f1

*pi* would generate the sequence

RV:*l* *p*
OFF *f1*

where the RV loads the value of *p,* given its block in the sub-opcode and offset in the following word, and the interpreter then adds the offset of the field *f1* in its record to get the correct address. OFF takes its argument in the sub-opcode if it is small enough.

**NIL**

The example above is incomplete, lacking a check for a **nil** pointer. The code generated would be

RV:*l* *p*
NIL
OFF *f1*

where the NIL operation checks for a *nil* pointer and generates the appropriate runtime error if it is.

**LVCON s,"**

A pointer to the specified length inline data is pushed onto the stack. This is primarily used for *printf* type strings used by WRITEF. (see sections 3.6 and 3.8)

**INX* s,w,w**

The operators INX2 and INX4 are used for subscripting. For example, the statement

a[i] := 2.0

with *i* an integer and *a* an "array [1..1000] of real" would generate

```
LV:l      a
RV4:l     i
INX4:8    1,999
CON8      2.0
AS8
```

Here the LV operation takes the address of *a* and places it on the stack. The value of *i* is then placed on top of this on the stack. The array address is indexed by the length 4 index (a length 2 index would use INX2) where the individual elements have a size of 8 bytes. The code for INX4 is:

```
_INX4:
        cvtbl   (lc)+,r0
        bneq    L1
        cvtwl   (lc)+,r0            #r0 has size of records
L1:
        cvtwl   (lc)+,r1            #r1 has lower bound
        movzwl  (lc)+,r2            #r2 has upper-lower bound
        subl3   r1,(sp)+,r3         #r3 has base subscript
        cmpl    r3,r2               #check for out of bounds
        bgtru   esubscr
        mull2   r0,r3               #calculate byte offset
        addl2   r3,(sp)             #calculate actual address
        jmp     (loop)
esubscr:
        movw    $ESUBSCR,_perrno
        jbr     error
```

Here the lower bound is subtracted, and range checked against the upper minus lower bound. The offset is then scaled to a byte offset into the array and added to the base address on the stack. Multi-dimension subscripts are translated as a sequence of single subscriptings.

**IND***

For indirect references through **var** parameters and pointers, the interpreter has a set of indirection operators that convert a pointer on the stack into a value on the stack from that address. different IND operators are necessary because of the possibility of different length operands. The IND14 and IND24 operators do conversions to long as they push their data.

### 2.7. Arithmetic operators

The interpreter has many arithmetic operators. All operators produce results long enough to prevent overflow unless the bounds of the base type are exceeded. The basic operators available are

Addition: ADD*, SUCC*
Subtraction: SUB*, PRED*
Multiplication: MUL*, SQR*
Division: DIV*, DVD*, MOD*
Unary: NEG*, ABS*

### 2.8. Range checking

The interpreter has several range checking operators. The important distinction among these operators is between values whose legal range begins at zero and those that do not begin at zero, for example a subrange variable whose values range from 45 to 70. For those that begin at zero, a simpler "logical" comparison against the upper bound suffices. For others, both the low and upper bounds must be checked independently, requiring two comparisons. On the VAX 11/780 both checks are done using a single index instruction so the only gain is in reducing the inline data.

## 2.9. Case operators

The interpreter includes three operators for **case** statements that are used depending on the width of the **case** label type. For each width, the structure of the case data is the same, and is represented in figure 2.4.

| |
|---|
| CASEOP |
| No. of cases |
| Case transfer table |
| Array of case label values |

Figure 2.4 – Case data structure

The CASEOP case statement operators do a sequential search through the case label values. If they find the label value, they take the corresponding entry from the transfer table and cause the interpreter to branch to the specified statement. If the specified label is not found, an error results.

The CASE operators take the number of cases as a sub-opcode if possible. Three different operators are needed to handle single byte, word, and long case transfer table values. For example, the CASEOP1 operator has the following code sequence:

```
_CASEOP1:
        cvtbl   (lc)+,r0
        bneq    L1
        cvtwl   (lc)+,r0            #r0 has length of case table
L1:
        movaw   (lc)[r0],r2         #r2 has pointer to case labels
        movzwl  (sp)+,r3            #r3 has the element to find
        locc    r3,r0,(r2)          #r0 has index of located element
        beql    caserr              #element not found
        mnegl   r0,r0               #calculate new lc
        cvtwl   (r2)[r0],r1         #r1 has lc offset
        addl2   r1,lc
        jmp     (loop)
caserr:
        movw    $ECASE,_perrno
        jbr     error
```

Here the interpreter first computes the address of the beginning of the case label value area by adding twice the number of case label values to the address of the transfer table, since the transfer table entries are 2 byte address offsets. It then searches through the label values, and generates an ECASE error if the label is not found. If the label is found, the index of the corresponding entry in the transfer table is extracted and that offset is added to the interpreter location counter.

## 2.10. Operations supporting pxp

The following operations are defined to do execution profiling.

**PXPBUF w**

Causes the interpreter to allocate a count buffer with *w* four byte counters and to clear them to zero. The count buffer is placed within an image of the *pmon.out* file as described in the *PXP Implementation Notes.* The contents of this buffer are written to the file *pmon.out* when the program ends.

**COUNT w**

Increments the counter specified by *w*.

**TRACNT w,A**

Used at the entry point to procedures and functions, combining a transfer to the entry point of the block with an incrementing of its entry count.

### 2.11. Set operations

The set operations: union ADDT, intersection MULT, element removal SUBT, and the set relationals RELT are straightforward. The following operations are more interesting.

**CARD s**

Takes the cardinality of a set of size *s* bytes on top of the stack, leaving a 2 byte integer count. CARD uses the **ffs** opcode to successively count the number of set bits in the set.

**CTTOT s,w,w**

Constructs a set. This operation requires a non-trivial amount of work, checking bounds and setting individual bits or ranges of bits. This operation sequence is slow, and motivates the presence of the operator INCT below. The arguments to CTTOT include the number of elements *s* in the constructed set, the lower and upper bounds of the set, the two *w* values, and a pair of values on the stack for each range in the set, single elements in constructed sets being duplicated with SDUP to form degenerate ranges.

**IN s,w,w**

The operator **in** for sets. The value *s* specifies the size of the set, the two *w* values the lower and upper bounds of the set. The value on the stack is checked to be in the set on the stack, and a Boolean value of *true* or *false* replaces the operands.

**INCT**

The operator **in** on a constructed set without constructing it. The left operand of **in** is on top of the stack followed by the number of pairs in the constructed set, and then the pairs themselves, all as single word integers. Pairs designate runs of values and single values are represented by a degenerate pair with both value equal. This operator is generated in grammatical constructs such as

**if** character **in** [`+´, ´−´, `*´, `/´]

or

**if** character **in** [`a´..`z´, `$´, `_´]

These constructs are common in Pascal, and INCT makes them run much faster in the interpreter, as if they were written as an efficient series of **if** statements.

### 2.12. Miscellaneous

Other miscellaneous operators that are present in the interpreter are ASRT that causes the program to end if the Boolean value on the stack is not *true,* and STOI, STOD, ITOD, and ITOS that convert between different length arithmetic operands for use in aligning the arguments in **procedure** and **function** calls, and with some untyped built-ins, such as SIN and COS.

Finally, if the program is run with the run-time testing disabled, there are special operators for **for** statements and special indexing operators for arrays that have individual element size that is a power of 2. The code can run significantly faster using these operators.

### 2.13. Mathematical Functions

The transcendental functions SIN, COS, ATAN, EXP, LN, SQRT, SEED, and RANDOM are taken from the standard UNIX mathematical package. These functions take double precision floating point values and return the same.

The functions EXPO, TRUNC, and ROUND take a double precision floating point number. EXPO returns an integer representing the machine representation of its argument's exponent, TRUNC returns the integer part of its argument, and ROUND returns the rounded integer part of its argument.

### 2.14. System functions and procedures

**LLIMIT**

A line limit and a file pointer are passed on the stack. If the limit is non-negative the line limit is set to the specified value, otherwise it is set to unlimited. The default is unlimited.

**STLIM**

A statement limit is passed on the stack. The statement limit is set as specified. The default is 500,000. No limit is enforced when the "p" option is disabled.

**CLCK**
**SCLCK**

CLCK returns the number of milliseconds of user time used by the program; SCLCK returns the number of milliseconds of system time used by the program.

**WCLCK**

The number of seconds since some predefined time is returned. Its primary usefulness is in determining elapsed time and in providing a unique time stamp.

The other system time procedures are DATE and TIME that copy an appropriate text string into a pascal string array. The function ARGC returns the number of command line arguments passed to the program. The procedure ARGV takes an index on the stack and copies the specified command line argument into a pascal string array.

### 2.15. Pascal procedures and functions

**PACK s,w,w,w**
**UNPACK s,w,w,w**

They function as a memory to memory move with several semantic checks. They do no "unpacking" or "packing" in the true sense as the interpreter supports no packed data types.

**NEW s**
**DISPOSE s**

An LV of a pointer is passed. NEW allocates a record of a specified size and puts a pointer to it into the pointer variable. DISPOSE deallocates the record pointed to by the pointer and sets the pointer to NIL.

The function CHR* converts a suitably small integer into an ascii character. Its primary purpose is to do a range check. The function ODD* returns *true* if its argument is odd and returns *false* if its argument is even.

The function UNDEF always returns the value *false*.

## 3. Input/output

### 3.1. The files structure

Each file in the Pascal environment is represented by a pointer to a *files* structure in the heap. At the location addressed by the pointer is the element in the file's window variable. Behind this window variable is information about the file, at the following offsets:

| | | |
|---|---|---|
| −108 | FNAME | Text name of associated UNIX file |
| −30 | LCOUNT | Current count of lines output |
| −26 | LLIMIT | Maximum number of lines permitted |
| −22 | FBUF | UNIX FILE pointer |
| −18 | FCHAIN | Chain to next file |
| −14 | FLEV | Pointer to associated file variable |
| −10 | PFNAME | Pointer to name of file for error messages |
| −6 | FUNIT | File status flags |
| −4 | FSIZE | Size of elements in the file |
| 0 | | File window element |

Here FBUF is a pointer to the system FILE block for the file. The standard system I/O library is used that provides block buffered input/output, with 1024 characters normally transferred at each read or write.

The files in the Pascal environment, are all linked together on a single file chain through the FCHAIN links. For each file the FLEV pointer gives its associated file variable. These are used to free files at block exit as described in section 3.3 below.

The FNAME and PFNAME give the associated file name for the file and the name to be used when printing error diagnostics respectively. Although these names are usually the same, *input* and *output* usually have no associated file name so the distinction is necessary.

The FUNIT word contains a set of flags. whose representations are:

| | | |
|---|---|---|
| EOF | 0x0100 | At end-of-file |
| EOLN | 0x0200 | At end-of-line (text files only) |
| SYNC | 0x0400 | File window is out of sync |
| TEMP | 0x0800 | File is temporary |
| FREAD | 0x1000 | File is open for reading |
| FWRITE | 0x2000 | File is open for writing |
| FTEXT | 0x4000 | File is a text file; process EOLN |
| FDEF | 0x8000 | File structure created, but file not opened |

The EOF and EOLN bits here reflect the associated built-in function values. TEMP specifies that the file has a generated temporary name and that it should therefore be removed when its block exits. FREAD and FWRITE specify that *reset* and *rewrite* respectively have been done on the file so that input or output operations can be done. FTEXT specifies the file is a text file so that EOLN processing should be done, with newline characters turned into blanks, etc.

The SYNC bit, when true, specifies that there is no usable image in the file buffer window. As discussed in the *Berkeley Pascal User's Manual,* the interactive environment necessitates having "input^" undefined at the beginning of execution so that a program may print a prompt before the user is required to type input. The SYNC bit implements this. When it is set, it specifies that the element in the window must be updated before it can be used. This is never done until necessary.

### 3.2. Initialization of files

All the variables in the Pascal runtime environment are cleared to zero on block entry. This is necessary for simple processing of files. If a file is unused, its pointer will be **nil.** All references to an inactive file are thus references through a **nil** pointer. If the Pascal system did not clear storage to zero before execution it would not be possible to detect inactive files in this simple way; it would probably be necessary to generate (possibly complicated) code to initialize each file on block entry.

When a file is first mentioned in a *reset* or *rewrite* call, a buffer of the form described above is associated with it, and the necessary information about the file is placed in this buffer. The file is also linked into the active file chain. This chain is kept sorted by block mark address, the FLEV entries.

### 3.3. Block exit

When block exit occurs the interpreter must free the files that are in use in the block and their associated buffers. This is simple and efficient because the files in the active file chain are sorted by increasing block mark address. This means that the files for the current block will be at the front of the chain. For each file that is no longer accessible the interpreter first flushes the files buffer if it is an output file. The interpreter then returns the file buffer and the files structure and window to the free space in the heap and removes the file from the active file chain.

### 3.4. Flushing

Flushing all the file buffers at abnormal termination, or on a call to the procedure *flush* or *message* is done by flushing each file on the file chain that has the FWRITE bit set in its flags word.

### 3.5. The active file

For input-output, *px* maintains a notion of an active file. Each operation that references a file makes the file it will be using the active file and then does its operation. A subtle point here is that one may do a procedure call to *write* that involves a call to a function that references another file, thereby destroying the active file set up before the *write.* Thus the active file is saved at block entry in the block mark and restored at block exit.†

### 3.6. File operations

Files in Pascal can be used in two distinct ways: as the object of *read, write, get,* and *put* calls, or indirectly as though they were pointers. The second use as pointers must be careful not to destroy the active file in a reference such as

write(output, input↑)

or the system would incorrectly write on the input device.

The fundamental operator related to the use of a file is FNIL. This takes the file variable, as a pointer, insures that the pointer is not **nil,** and also that a usable image is in the file window, by forcing the SYNC bit to be cleared.

A simple example that demonstrates the use of the file operators is given by

writeln(f)

that produces

```
RV:l     f
UNIT
WRITLN
```

### 3.7. Read operations

**GET**

Advance the active file to the next input element.

---

† It would probably be better to dispense with the notion of active file and use another mechanism that did not involve extra overhead on each procedure and function call.

**FNIL**

A file pointer is on the stack. Insure that the associated file is active and that the file is synced so that there is input available in the window.

**READ***

If the file is a text file, read a block of text and convert it to the internal type of the specified operand. If the file is not a text file then do an unformatted read of the next record. The procedure READLN reads upto and including the next end of line character.

**READE A**

The operator READE reads a string name of an enumerated type and converts it to its internal value. READE takes a pointer to a data structure as shown in figure 3.2.

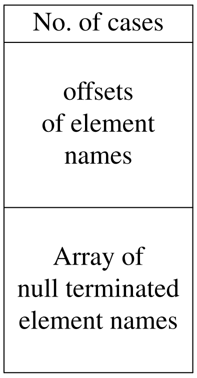


Figure 3.2 – Enumerated type conversion structure

See the description of NAM in the next section for an example.

### 3.8. Write operations

**PUT**

Output the element in the active file window.

**WRITEF s**

The argument(s) on the stack are output by the *fprintf* standard I/O library routine. The sub-opcode *s* specifies the number of longword arguments on the stack.

**WRITEC**

The character on the top of the stack is output without formatting. Formatted characters must be output with WRITEF.

**WRITES**

The string specified by the pointer on the top of the stack is output by the *fwrite* standard I/O library routine. All characters including nulls are printed.

**WRITLN**

A linefeed is output to the active file. The line-count for the file is incremented and checked against the line limit.

**PAGE**

A formfeed is output to the active file.

**NAM A**

The value on the top of the stack is converted to a pointer to an enumerated type string name. The address A points to an enumerated type structure identical to that used by READE. An error is raised if the value is out of range. The form of this structure for the predefined type **boolean** is shown in figure 3.3.

| | |
|---|---|
| *bool*: | 2 |
| | 6 |
| | 12 |
| | 17 |
| | "false" |
| | "true" |

Figure 3.3 – Boolean type conversion structure

The code for NAM is

```
_NAM:
        incl     lc
        addl3    (lc)+,ap,r6          #r6 points to scalar name list
        movl     (sp)+,r3             #r3 has data value
        cmpw     r3,(r6)+             #check value out of bounds
        bgequ    enamrng
        movzwl   (r6)[r3],r4          #r4 has string index
        pushab   (r6)[r4]             #push string pointer
        jmp      (loop)
enamrng:
        movw     $ENAMRNG,_perrno
        jbr      error
```

The address of the table is calculated by adding the base address of the interpreter code, *ap* to the offset pointed to by *lc*. The first word of the table gives the number of records and provides a range check of the data to be output. The pointer is then calculated as

```
tblbase = ap + A;
size = *tblbase++;
return(tblbase + tblbase[value]);
```

**MAX s,w**

The sub-opcode *s* is subtracted from the integer on the top of the stack. The maximum of the result and the second argument, *w*, replaces the value on the top of the stack. This function verifies that variable specified width arguments are non-negative, and meet certain minimum width requirements.

**MIN s**

The minimum of the value on the top of the stack and the sub-opcode replaces the value on the top of the stack.

The uses of files and the file operations are summarized in an example which outputs a real variable (r) with a variable width field (i).

```
writeln(´r =´,r:i,´ ´,true);
```

that generates the code

```
UNITOUT
FILE
CON14:1
CON14:3
LVCON:4     "r ="
WRITES
RV8:l       r
RV4:l       i
MAX:8       1
RV4:l       i
MAX:1       1
LVCON:8     " %*.*E"
FILE
WRITEF:6
CONC4       ´ ´
WRITEC
CON14:1
NAM         bool
LVCON:4     "%s"
FILE
WRITEF:3
WRITLN
```

Here the operator UNITOUT is an abbreviated form of the operator UNIT that is used when the file to be made active is *output*. A file descriptor, record count, string size, and a pointer to the constant string "r =" are pushed and then output by WRITES. Next the value of *r* is pushed on the stack and the precision size is calculated by taking seven less than the width, but not less than one. This is followed by the width that is reduced by one to leave space for the required leading blank. If the width is too narrow, it is expanded by *fprintf*. A pointer to the format string is pushed followed by a file descriptor and the operator WRITEF that prints out *r*. The value of six on WRITEF comes from two longs for *r* and a long each for the precision, width, format string pointer, and file descriptor. The operator CONC4 pushes the *blank* character onto a long on the stack that is then printed out by WRITEC. The internal representation for *true* is pushed as a long onto the stack and is then replaced by a pointer to the string "true" by the operator NAM using the table *bool* for conversion. This string is output by the operator WRITEF using the format string "%s". Finally the operator WRITLN appends a newline to the file.

### 3.9. File activation and status operations

**UNIT***

The file pointed to by the file pointer on the top of the stack is converted to be the active file. The opcodes UNITINP and UNITOUT imply standard input and output respectively instead of explicitly pushing their file pointers.

**FILE**

The standard I/O library file descriptor associated with the active file is pushed onto the stack.

**EOF**

The file pointed to by the file pointer on the top of the stack is checked for end of file. A boolean is returned with *true* indicating the end of file condition.

**EOLN**

The file pointed to by the file pointer on the top of the stack is checked for end of line. A boolean is returned with *true* indicating the end of line condition. Note that only text files can check for end of line.

### 3.10. File housekeeping operations

**DEFNAME**

Four data items are passed on the stack; the size of the data type associated with the file, the maximum size of the file name, a pointer to the file name, and a pointer to the file variable. A file record is created with the specified window size and the file variable set to point to it. The file is marked as defined but not opened. This allows **program** statement association of file names with file variables before their use by a RESET or a REWRITE.

**BUFF s**

The sub-opcode is placed in the external variable *_bufopt* to specify the amount of I/O buffering that is desired. The current options are:

0 – character at a time buffering
1 – line at a time buffering
2 – block buffering

The default value is 1.

**RESET**
**REWRITE**

Four data items are passed on the stack; the size of the data type associated with the file, the maximum size of the name (possibly zero), a pointer to the file name (possibly null), and a pointer to the file variable. If the file has never existed it is created as in DEFNAME. If no file name is specified and no previous name exists (for example one created by DEFNAME ) then a system temporary name is created. RESET then opens the file for input, while REWRITE opens the file for output.

The three remaining file operations are FLUSH that flushes the active file, REMOVE that takes the pointer to a file name and removes the specified file, and MESSAGE that flushes all the output files and sets the standard error file to be the active file.

## 4. Conclusions

It is appropriate to consider, given the amount of time invested in rewriting the interpreter, whether the time was well spent, or whether a code-generator could have been written with an equivalent amount of effort. The Berkeley Pascal system is being modified to interface to the code generator of the portable C compiler with not much more work than was involved in rewritting *px*. However this compiler will probably not supercede the interpreter in an instructional environment as the necessary loading and assembly processes will slow the compilation process to a noticeable degree. This effect will be further exaggerated because student users spend more time in compilation than in execution. Measurements over the course of a quarter at Berkeley with a mixture of students from beginning programming to upper division compiler construction show that the amount of time in compilation exceeds the amount of time spent in the interpreter, the ratio being approximately 60/40.

A more promising approach might have been a throw-away code generator such as was done for the WATFIV system. However the addition of high-quality post-mortem and interactive debugging facilities become much more difficult to provide than in the interpreter environment.