SETL Newsletter # 161

Art Grand
December 31, 1975

Specifications for a new optimizer-oriented SETL front end

This newsletter specifies the design for a new SETL front end. Its output is a set of quadruples which can either be interpreted or used as input to the SETL optimizer.

The front end follows the general scheme outlined in LITTLE Newsletter 40. It uses a standard scanner-parser to produce a reverse polish string in which arguments are represented as symbol table pointers and operations as calls to semantic routines. The polish string is processed by a semantic pass whose output is a set of quadruples. In the initial implementation these quadruples will be interpreted directly. Later we will insert the optimizer between the semantic processor and interpreter. In a final implementation we may convert the quadruples into the polish II form described in newsletter 40, and from there to machine code.

As described in newsletter 40, the parser passes 4 tables to the semantic pass: *Symtab*, the symbol table, *Names*, an array of token names, *Values* containing values of constants, and *polish1*, the polish string. The semantic processor produces an augumented *Symtab*, *Names* and the set of quadruples *Code*. It also contains conditional code to produce *Routab* and *Blocktab* containing information on subroutines and basic blocks needed by the optimizer. The front end will gather statistics as to frequency of operations, etc., to facilitate data structure choice in the optimizer.

Data Structures

We now present detailed specifications of the above mentioned data structures.

Notice that they are all LITTLE bit string arrays, thus implementation of the front end need not await the final debugging of MIDL.

SYMTAB

Symtab is a variation of the general purpose symbol table presented in newsletter 40. As before we provide certain standard fields and fields whose use varies from pass to pass however the number of standardized fields is less than previously described. Symtab requires hashing only during parsing. During the remainder of compilation it is desirable to have symtab as compact as possible, without the gaps normally associated with hashing. We do this with a linked list algorithm which chains together entries with the same hash code. The head of each collision chain is stored in an auxiliary hash table which is dropped after parsing.

Symtab has the following fields:

STYPE: Lexical type

SNAME: Pointer to the *names* array in which token names are stored.

SLENGTH: Length of the token.

SSCOPE: Scope of identifier. This field is set during semantic processing

SPARAM: Parameter number, determinedly semantic processor.

SVAL: Pointer to *Val* for value of constant.

SLINK: This field is used differently during each pass. During parsing it links different names with the same hash code. In the semantic pass it links variables of the same name in different scopes. In the optimizer it points to the global attribute table ATAB.

SFLAG: Flags procedure and label constants.

SDEF: This field gives the block in which procedure and label constants are defined.

SNUM: This field overlaps SDEF and gives the number of o and i occurrences of a variable.

SOFFSET: This overlaps SDEF and SNUM and is used by the interpreter for storage allocation.

SLOCAL: Flags local variables.

NAMES

The *names* table contains the names of tokens represent as packed characters.

VAL

This array stores the internal values of constants in their LITTLE representation.

POLISH

The polish string will be implemented as a packed array. Names and constants are represented as *symtab* pointers. Counters and marker nodes are integers biased by the dimension of symtab.

CODE

Code represents the program as an array of tuples. Each operation is represented by a 'root' entry which contains its opcode, output variable and first z input variables. Additional inputs are stored in successive entries.

The fields of code items are:

OVAR: The output variable.

IVAR1: First input.

IVAR2: Second input.

NARGS: Number of inputs.

OPCODE: Operation code.

OPTYPE: Extra type information generated by the optimizer, corresponding to the *etype* field of run time objects.

Routab

This is an auxilliary table created for the optimizer. It contains the following information about each routine:

ROUTNAME: Pointer to *symtab*, for routine's name.

ROUTENTRY: Number of routine's entry block.

ROUTEXIT: Number of its exit block.

ROUTARGS: Number of formal parameters.

BLOCKTAB

This is another auxilliary table giving information about each basic block. Its fields are:

BSTART:	Start of block in *code*.
BLEN:	Length of block.
BCESS:	Pointer to optimizer's cessor map.
BNCESS:	Number of cessors.
BPRED:	Pointer to predecassor map.
BNPRED:	Number of predecessors.

THE QUADRUPLE LANGUAGE

The quadruples produced by the semantic pass serve two purposes: first these will function as primitives for the interpreter; second they will serve as input to the optimizer. These goals are not always compatible. In many cases the interpreter will require more detailed primitives than the optimizer and vice versa. As a result certain operations are only used by one or the other.

The following is a list of quadruple of codes:

Binary Operations

odv	division
omxm	maximum
omnm	minimum
oad	addition
osb	subtraction
omult	multiplication
oor	or
oand	and
oxor	exclusive or
orm	remainder

Relational Operations

oeq	equal
one	not equal
ole	less than or equal
olt	less than
ogt	greater than
oge	greater or equal
oelm	membership test
oinc	inclusion test
oimp	implication

Unary Operations

oabs	obsolute value
osiz	number of elements
onot	logical not
ohd	head
otl	tail
oarb	arbitrary element
ooct	octal conversion
odec	decimal conversion
otype	type

Operations on Sets

oset	set former
opw	powerset
onpw	n-power set
owth	with
olss	less
olsf	lessf

Operations on Tuples and strings

otpl	tuple former
osub	s(i:j)
oend	s(i:)

Mappings and Function Evaluations

oof	f(x)	and $f(x_1 \ldots x_n)$
oofa	f{x}	and $f\{x_1 \ldots x_n\}$
oofb	f[x]	and $f[x_1 \ldots x_n]$
oindex	f(x)	where x is known to be a tuple.
ofcall	f(x)	where x is known to be a function.

Assignments

oass	simple assignment
argin	argument-in assignment
argout	argument-out assignment
osof	f(x) = y and $f(x_1 \ldots x_n) = y$
osof	f{x} = y and $f\{x_1 \ldots x_n\} = y$
osofb	f[x] = y and $f[x_1 \ldots x_n] = y$
osubs	s(I:j) = y
oretasin	return assignment
osend	s(I:) = y
oindexs	s(I) = y where s *is* a tuple

Control Statements

ocall	subroutine call
ogoto	go to *ivar2*
oifgo	if *ivar1* then goto *ivar2*
oifnot	if <u>not</u> *ivar1* then goto *ivar2*
oretgo	return
oexit	program exit
onext	set theoretic iterator
onextq	as onext, but as part of quantifier expression
onexts	as onext, but as part of set former.
oinit	branch to *ivar2* if this is not the first call to the current routine

Input - Output

ofile	makefile
oprint	print
owrite	write
oread	read

Miscellaneous

onew	newat

Pseudo Operations

These operations have no run-time semantics and are used only to simplify value flow analysis.

auxarb	dummy arb operation
auxset	dummy set former
auxass	dummy assignment
auxtl	a pseudo operation used to aid the analysis of mappings. It has the semantics:

```
definef auxtl x;
return if type x ne tupl then orm
       else if # x gt 2 then tl x
       else x(2);
end auxtl;
```

This operation is designed to return the same values for <1,2,3> and <1, <2,3>>.

auxoralt	declares two variables to have the same value.
auxnonull	declares its argument to be a non-null set. not containing its first element.
auxissing	declares a single valued mapping.
auxlab	defines a label
auxsub	defines a subroutine.

Most of the above operations have an obvious meaning. We concentrate on the more complicated cases. We denote quadruples by the notation <*output, input1,...,inputn*>

General program structure

Each routine will have compiler generated entry and exit blocks containing auxiliary assignments used for value flow analysis. Functions return their values by means of the operation

<temp, retasin, value>

The optimizer will view this as an

assignment to a temporary, while the interpreter will view it as an assignment to a special global variable, register, etc.

The following is a typical entry block for an n-parameter procedure named *sub:*

<sub, auxsub >
<p_1, auxass, p_1>
.
.
.
<p_n, auxass, p_n>

where p_1 through p_n are the formal parameters. The corresponding exit block will be

<elab, auxlab>
<p_1, auxass, p_1>
<p_n, auxass, p_n>

where *elab* is a compiler generated label. A return statement will be translated as

<oretgo, elab>

Oretgo is treated as a goto by the optimizer and a return by the interpreter. For functions we generate a temporary *restemp* to store the result. The exitblock is then compiled as

<elab, deflab>
<p_1, auxass, p_1>
.
.
.
<p_n, auxass, p_n>
<restemp, auxass, restemp>

The extra auxass is used to chain the function value to its uses. The statement

return x;

is translated as

<restemp, retassin, x>
< , oretgo, elab>

Subroutine calls

Subroutine calls are preceded by *argument in* assignments and followed by *argument out* assignments. *Argument in* assignments are always generated by the semantic processor. *Argument out* assignments are generated when the arguments are program variables as opposed to temporaries.

If an argument is an extraction operator, such as <u>hd</u> x we generate an *argument out* assignment for the temporary which holds <u>hd</u> x and then a sinister assignment to copy the temporary into x. General sinister assignments recieve more complex treatment.

The statement

$$f(x_1,\ldots,x_n);$$

expands to:

$$\begin{array}{l}
\langle t_1,\ \text{argin},\quad x_1\rangle \\
\qquad \vdots \\
\langle t_n,\ \text{argin},\quad x_n\rangle \\
\langle\ \ ,\ \text{ocall},\ f,\ t_1,\ t_z,\ldots,t_n\rangle \\
\langle x_1,\ \text{argout},\quad t_1\rangle \\
\qquad \vdots \\
\langle x_n,\ \text{argout},\quad t_n\rangle
\end{array}$$

Function Calls and Mappings

Function calls and mappings are indistingiushable at compile time. Whenever we encounter expressions such as $y = f(x_i,\ldots,x_n)$ we generate code to treat f as both a function and a mapping. The primitives *oof oofa*, and *oofb* are assumed to include run time tests of these arguments.

We generate a complex series of *aux* operators prior to each function call to provide proper analysis of mappings. The *auxtl* operator is emitted once for each argument to simulate a series of single argument mappings. In addition we generate the *argument in* and *argument out* assignments used in subroutine calls and eliminate them if the optimizer determines that an expression is a mapping rather than a function call.

For $y = f(x_1,\ldots,x_n)$ we generate

<t_1, arginass, x_1>
<t_n, arginass, x_n>
(1) <temp, auxarb, f>
<temp, auxtl, temp> /* we emit this instruction
⋮ *n* times */
(2) <y, oof, $t_1,\ldots t_n$, temp>
<x_1, argoutass, t_1>
<x_n, argoutass t_n>.

Since f{x} can only represent a mapping, we generate only line (1) through (2) with *oofa* substituted for *oof*. For f[x] we generate the same pattern as f(x) but omit the argument out assignments.

Sinister Assignments

Below we list the code generated for primitive sinister assignments. General sinister assignments are compiled into combinations of these plus the extraction primitives.

Source Form	Quadruple Form
$f(x_1,\ldots,x_n) = y$	<t, otpl, $x_1,\ldots,x_n$, y> <f, osof, f, t>
$f\{x_1,\ldots,x_n\} = y$	<t, otpl, $x_1,\ldots,x_n$, y> <f, osofa, f, t>
$f[x_1,\ldots,x_n] = y$	<t_1,..., auxarb, x_1> <t_n, auxarb, x_n> <t_{n+1} auxarb, y> <temp, auxtup, $t_1,\ldots,t_{n+1}$> <$temp_2$, otpl, $x_1,\ldots,x_n$, y> <f, osofb, f, $temp_2$, temp>

```
S(I:) = y                   <S, oends, S, I, y>
S(I: j) = y                 <S, osubs, S, I, j, y>
```

Multiple assignments are treated as individual indexed and subtuple assignments. However, in generating

y = s(1)

as part of a multiple assignment we use the oindex primitive instead of the oof primitive.

Set Iterators

The code for the loop

```
(∀ x ε s)
block
end ∀;
```

is the most complex code fragment we emit. We show it first without the auxilliary operators required by the optimizers.

```
<temp, oass, o> /* temp is an auxilliary pointer
                   used by SRTL                  */
<test, auxlab>
<x, next, s, temp>
<templ, oeq, x, om>
<     , oifgo, templ, elab>
      block
<     , ogoto, test>
<elab, auxlab>
```

With the auxilliary operators added we have:

```
<temp, oass, o>
<s1, auxass, nl>
<s2, auxass, nl>
<test, auxlab,    >
<x, onext, s temp>
<templ, oeq, x, om>
<temp2, oeq, s1, s>
<temp3, auxoralt, templ, temp2>
<     , oifgo, temp3, elab>
```

```
<temp4, ominus, s, s1>
<temp5, auxarb, temp4>
<temp6, auxoralt, x, temp5>
<x, auxass, temp6>
<temp7, auxset, x>
<temp8, oadd, s1, temp7>
<s1, auxass, temp8>
     block
<temp9, auxset, x >
<temp10, oadd, s2, temp9>
<temp11, auxoralt, temp10, s1>
<s2, auxass, temp11>
<     , ogoto, test>
<elab, auxlab>
<temp12, auxoralt, s1, s>
<s, auxass, temp12>
```

This reflects the code sequence contained in *Optimization of Very High Level Languages II.*