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 augmented 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 implementa-
tion 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 represented 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 auxiliary 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.
ROUDEXIT: Number of its exit block.
ROUTARGS: Number of formal parameters.
This is another auxiliary 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 predecessor map.
- **BNPRED**: Number of predecessors.

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
- omnmm: 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: absolute value
- osiz: number of elements
- onot: logical not
- ohd: head
- otl: tail
- oarb: arbitrary element
- coct: 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

\begin{align*}
oof & \quad f(x) \quad \text{and} \quad f(x_1 \ldots x_n) \\
oofa & \quad f(x) \quad \text{and} \quad f(x_1 \ldots x_n) \\
oofb & \quad f(x) \quad \text{and} \quad f(x_1 \ldots x_n) \\
index & \quad f(x) \quad \text{where } x \text{ is known to be a tuple.} \\
ocal & \quad f(x) \quad \text{where } x \text{ is known to be a function.}
\end{align*}

Assignments

\begin{align*}
\textrm{assin} & \quad \text{simple assignment} \\
\argin & \quad \text{argument-in assignment} \\
\argout & \quad \text{argument-out assignment} \\
\osof & \quad f(x) = y \quad \text{and} \quad f(x_1 \ldots x_n) = y \\
\osofa & \quad f(x) = y \quad \text{and} \quad f(x_1 \ldots x_n) = y \\
\osofb & \quad f(x) = y \quad \text{and} \quad f(x_1 \ldots x_n) = y \\
\osubs & \quad s(I:j) = y \\
\oretasin & \quad \text{return assignment} \\
\osend & \quad s(I:) = y \\
\oindexs & \quad s(I) = y \quad \text{where } s \text{ is a tuple}
\end{align*}

Control Statements

\begin{align*}
\ocal & \quad \text{subroutine call} \\
\ogoto & \quad \text{go to } i\text{var2} \\
\oi & \quad \text{if } i\text{var1} \text{ then goto } i\text{var2} \\
\oinot & \quad \text{if not } i\text{var1} \text{ then goto } i\text{var2} \\
\oretgo & \quad \text{return} \\
\oexit & \quad \text{program exit} \\
\onext & \quad \text{set theoretic iterator} \\
\onextq & \quad \text{as onext, but as part of quantifier expression} \\
\onexts & \quad \text{as onext, but as part of set former.} \\
\oinit & \quad \text{branch to } i\text{var2} \text{ if this is not the first call to} \\
& \quad \text{the current routine}
\end{align*}
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:

\[
\text{define auxtl } x; \\
\text{return if } \text{type } x \neq \text{tupl then } \text{orm} \\
\text{else if } \# x' \geq 2 \text{ then } \text{tl } x \\
\text{else } x(2); \\
\text{end auxtl;}
\]

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

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 \(<output, input_1, \ldots, input_n>\)

**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

\(<\text{temp}, \text{retasin}, \text{value}>\)

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:

\[
\langle \text{sub, auxsub} \rangle
\]
\[
\langle p_1, \text{auxass}, p_1 \rangle
\]
\[
\quad \vdots
\]
\[
\langle p_n, \text{auxass}, p_n \rangle
\]

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

\[
\langle \text{elab, auxlab} \rangle
\]
\[
\langle p_1, \text{auxass}, p_1 \rangle
\]
\[
\langle p_n, \text{auxass}, p_n \rangle
\]

where \( \text{elab} \) is a compiler generated label. A return statement will be translated as

\[
\langle \text{oretgo, elab} \rangle
\]

\( \text{oretgo} \) is treated as a goto by the optimizer and a return by the interpreter. For functions we generate a temporary \( \text{restemp} \) to store the result. The exit block is then compiled as

\[
\langle \text{elab, deflab} \rangle
\]
\[
\langle p_1, \text{auxass}, p_1 \rangle
\]
\[
\quad \vdots
\]
\[
\langle p_n, \text{auxass}, p_n \rangle
\]
\[
\langle \text{restemp, auxass, restemp} \rangle
\]

The extra \( \text{auxass} \) is used to chain the function value to its uses. The statement

\[
\text{return } x;
\]

is translated as

\[
\langle \text{restemp, retassin, x} \rangle
\]
\[
\langle \text{oretgo, elab} \rangle
\]
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 \( \text{hd } x \) we generate an *argument out* assignment for the temporary which holds \( \text{hd } x \) and then a sinister assignment to copy the temporary into \( x \). General sinister assignments receive more complex treatment.

The statement

\[ f(x_1, \ldots, x_n); \]

expands to:

\[
< t_1, \text{argin}, \ x_1 > \\
\vdots \\
< t_n, \text{argin}, \ x_n > \\
< \ , \text{ocall}, f, t_1, t_2, \ldots, t_n > \\
< x_1, \text{argout}, \ t_1 > \\
\vdots \\
< x_n, \text{argout}, \ t_n >
\]

Function Calls and Mappings

Function calls and mappings are indistinguishable at compile time. Whenever we encounter expressions such as \( y = f(x_1, \ldots, x_n) \) 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

1. \(<t_1, \text{arginass}, x_1>\)
2. \(<t_n, \text{arginass}, x_n>\)
3. \(<\text{temp}, \text{auxarb}, f>\)
4. \(<\text{temp}, \text{auxtl}, \text{temp} /* we emit this instruction */ n \text{ times}>\)

\[ \langle y, \text{oof}, t_1, \ldots t_n, \text{temp} \rangle \]
\[ \langle x_1, \text{argoutass}, t_1 \rangle \]
\[ \langle x_n, \text{argoutass}, t_n \rangle . \]

Since \( f(x) \) can only represent a mapping, we generate only line (1) through (2) with \( \text{oof} a \) substituted for \( \text{oof} \). 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.

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Quadruple Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(x_1, \ldots, x_n) = y )</td>
<td>(&lt;t, \text{otpl}, x_1, \ldots, x_n, y&gt;)</td>
</tr>
<tr>
<td></td>
<td>(&lt;f, \text{osof}, f, t&gt;)</td>
</tr>
<tr>
<td>( f{x_1, \ldots, x_n} = y )</td>
<td>(&lt;t, \text{otpl}, x_1, \ldots, x_n, y&gt;)</td>
</tr>
<tr>
<td></td>
<td>(&lt;f, \text{osof}, f, t&gt;)</td>
</tr>
<tr>
<td>( f[x_1, \ldots, x_n] = y )</td>
<td>(&lt;t_1, \ldots, \text{auxarb}, x_1&gt;)</td>
</tr>
<tr>
<td></td>
<td>(&lt;t_n, \text{auxarb}, x_n&gt;)</td>
</tr>
<tr>
<td></td>
<td>(&lt;t_{n+1} \text{auxarb}, y&gt;)</td>
</tr>
<tr>
<td></td>
<td>(&lt;\text{temp}, \text{auxtp}, t_1, \ldots, t_{n+1}&gt;)</td>
</tr>
<tr>
<td></td>
<td>(&lt;\text{temp}_2, \text{otpl}, x_1, \ldots, x_n, y&gt;)</td>
</tr>
<tr>
<td></td>
<td>(&lt;f, \text{osofb}, f, \text{temp}_2, \text{temp}&gt;)</td>
</tr>
</tbody>
</table>
SETL-161-12

\[ S(I:) = y \quad \langle S, \text{oends}, S, I, y \rangle \]
\[ S(I: j) = y \quad \langle S, \text{osubs}, S, I, j, y \rangle \]

Multiple assignments are treated as individual indexed and subtuple assignments. However, in generating
\[ y = s(l) \]
as part of a multiple assignment we use the oindex primitive instead of the oof primitive.

Set Iterators

The code for the loop
\[
(\forall x \in s)
\]
\[ blook \]
\[ \text{end } \forall; \]
is the most complex code fragment we emit. We show it first without the auxiliary operators required by the optimizers.

\[ \langle \text{temp}, \text{oass}, o \rangle /* \text{temp is an auxiliary pointer used by SRTL} */ \]
\[ \langle \text{test}, \text{auxlab} \rangle \]
\[ \langle x, \text{next}, s, \text{temp} \rangle \]
\[ \langle \text{templ, oeq, x, om} \rangle \]
\[ \langle , \text{oifgo, templ, elab} \rangle \]
\[ blook \]
\[ \langle , \text{ogoto, test} \rangle \]
\[ \langle \text{elab, auxlab} \rangle \]

With the auxiliary operators added we have:
\[ \langle \text{temp, oass, o} \rangle \]
\[ \langle s_1, \text{auxass, ni} \rangle \]
\[ \langle s_2, \text{auxass, ni} \rangle \]
\[ \langle \text{test, auxlab, } \rangle \]
\[ \langle x, \text{cnext, s temp} \rangle \]
\[ \langle \text{templ, oeq, x, om} \rangle \]
\[ \langle \text{temp2, oeq, s}_{1}, s \rangle \]
\[ \langle \text{temp3, auxoralt, templ, temp2} \rangle \]
\[ \langle , \text{oifgo, temp3, elab} \rangle \]
<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*. 