LITTLE Newsletter # 40

Art Grand
November 26, 1975

Standardised and More Efficient Communication with the LITTLE Codegenerators.

The SETL group has begun to support a variety of translators which have similar requirements for lexical scanners etc. More recent compilers have attempted to reuse portions of their predecessors. Adaptations of existing code have been rather ad hoc. For example, both MIDL and SETL produce LITTLE as intermediate text and then use the entire LITTLE compiler as a code generator. Clearly to achieve more efficient translation, we must either write individual code generators for each language, or design a standard code generator which can be used for all of them. In this Newsletter we explore the latter approach, and as an illustration of it outline a modular compiler design for MIDL which will allow two-thirds of its code to be reused efficiently and elegantly by other languages.

The widest variations found in the translators implemented so far lie in their semantic processing routines and in their choice of data structures. The simulaneous parsing semantic-processing scheme used by LITTLE makes it awkard to reuse the very flexible parser which LITTLE incorporates. The assortment of tables passed to the LITTLE code generator are so hard to describe and duplicate that other compilers can only communicate with it via source code.

By contrast, the new compiler design sketched here will make a clear distinction between parsing and semantic processing. The data structures passed between parser and semantic processor will be polish strings, a structure defined by a formal grammar. This will facilitates the combination of 'custom' compiler sections, written for a particular language, with 'general purpose ' sections which work effectively for many languages.

The compiler to be outlined will have four passes. The first will use the LITTLE lexical scanner. The second pass will be a table driven parser which produces a polish string whose operators correspond to the semantic routines of the language. A third, semantic pass will produce 'VOA-like' operators which are close to machine language. This string will then be processed by a standard code generator. In the following sections, we present a MIDL parser and semantic processor of the type envisaged. Then we conclude with a brief discussion of code generation and of the adaptation of our scheme to other languages.

Parsing

The parser we propose to use is a modified version of the LITTLE parser. As source-text symbols are recognized, our parser reorders and standardises them and writes them onto the first polish string (which we call the polish I string). Unlike the current parser, which immediately calls generator routines, the new parser will merely write special 'marker' nodes onto the string as necessary. These will subsequently be recognized by the semantic pass, during which the generators will actually be called.

The parser will not require the user to write his own auxiliary routines, but merely to supply an appropriate grammar. To facilitate the writing of parsers of the type envisaged, we propose to incorporate several primitives, such as branch on literals and operator precedence parsing of expressions into the meta-language which defines them. Our extended meta-language will use the following notations.

SYMBOL	FUNCTION
< * TP >	Find a token of lexical type TP, hash it into the symbol table, and write a pointer to it onto polish I.
< CL >	Find a clause CL
< CL * N >	Find at least N repititions of clause CL and write the repitition count onto polish I.
OP	Write the marker node OP onto polish I.
BRONLIT 'LITI' :LABI,...'LITN': LABN	If next token is the literal 'LITi' branch to LABi.
[STARTEXP]	Initialize the precedence parser.
[ENDEXP]	'Clean up' after precedence parse.
BINPREC '+' :3:1, '-':3:2	Parses binary operators. Each argument is a triple <literal, precedence, opcode>. When an operator is emitted, opcode BINOP is written onto polish I.
UNPREC 'NOT.' :6:7	Parses unary operators.
[ALLRI]	Set parser success flag to true.
[OPEN(TP)]	Indicates that a compound statement of type TP has been opened. TP is pushed onto a stack.
[END]	Pops the stack and checks that an end statement matches its opener.
[ER(N)]	Print error message N.
.B	Return failure
.	Go to next alternate production.
.L	Go to label L on failure.
.L(N)	As above, but set error number to N.
..L	Go to label L.
..L(N)	As above, but set error number.

M I D L   G R A M M A R

THE FOLLOWING IS THE GRAMMAR FOR MIDL. MANY ERROR MESSAGES HAVE BEEN OMITTED FOR SAKE OF CLARITY. IN THE KEYPUNCHED VERSION, MARKER NODES ARE WRITTEN:
-NODE- .

```
<STATEME>     → SUBR <*NAME> ≠(≠ <*NAME> <CNAME*0>≠)≠ <RWORD> -SUBR1-/
                       ,ER       ,OKSUB      ,ER       ,ER
              → FNCT <*NAME> ≠(≠ <*NAME> <CNAME*0> ≠)≠ <RWORD> -SUBR3-/
                   .ER      ,ER      ,ER               ,ER
              →  <LAB>   ,,UNLAB
              →  <UNLAB>

<CNAME>       →  ≠,≠  <NAME> / ,B ,ER

<RWORD>       →  ≠RECURSIVE≠ -RECURSIVE- /,
              →  [ALLRI]

<OKSUB>       →  <RWORD> -SUBR2-

<LAB>         →  ≠/≠  <*NAME>  ≠/≠ -LABEL- /,B .ER  ,ER

<UNLAB>       →    BRONLIT
                   ≠IF≠ : IFS,
                   ≠WHILE≠: WHILS,
                   ≠UNTIL≠:UNTLS,
                   ≠DO≠ : DOS,
                   ≠END≠ : END,
                   ≠ELSE≠ : ELSE,
                   ≠SIZE≠ : SIZS,
                   ≠REAL≠ : RLS,
                   ≠DCL≠ : DCLS,
                   ≠EXPECT≠:EXPECTS,
                   ≠TYP≠ : TYPS,
                   ≠DATA≠ : DATAS,
                   ≠NAMESET≠: NMS,
                   ≠ACCESS≠ : ACS,
                   ≠FOR≠   : FOR /,
              →  <SIMPST>

  <IFS>       →  <EXP> ≠THEN≠ [OPEN(IFTHEN)] -IFT-,. STATEME/
              →  ≠(≠ <EXP>  ≠)≠ -SIMPLEF1- <SIMPST> -SIMPLIF2-  ,,REST/
```

```
<WHILS>       →   [OPEN(WHILE)] -WHILE1-  <EXP> -WHILE2-   ,,REST/

<UNTLS>       →   [OPEN(UNTIL)] -UNTIL1-  <EXP> -UNTIL2-   ,,REST/

<DOS>         →   [OPEN(DO)] <*NAME> -DO1- ≠=≠ <EXP> -DO2-  ≠TO≠ <EXP> -DO3-
                  <BYPART> -DO4-   ,,REST/
                         ,ER       ,ER    ,ER           ,ER   ,ER   ,
              → /*NO  -BYPART- */ -DO5-   .,REST

<BYPART>      →   ≠BY≠ ≠-≠ <EXP> -DOWN- /,B   ,    ,ER
              →   <EXP>/ ,B

<END>         →   [END] -END-  [ALLRI]/

<ELSE>        →   -ELSE-    ,,STATEME /

<SIZE>        →   <ATTRSP>  <CATTRSP*0) <STRWORD> -SIZE-   ,,REST/

<RLS>         →   <*NAME>  <CNAME *0> <STRWORD> -REAL-   ,,REST/

<STRWORD>     →   ≠STACK≠   -STACK-/,
              →   ≠NOSTACK≠ -NOSTACK- /,
              →   [ALLRI]

<ATTRSP>      →   <*NAME>  ≠(≠ <IEXP> ≠)≠ /

<CATTRSP>     →   ≠,≠ <ATTRSP>  / ,B ,ER

<DCLS>        →   <VARDCL> <CVARDCL*0> <STRWORD> -DCL-

<VARDCL>      →   <*NAME>  <TYPEDES>/

<CVARDCL>     →   ≠,≠ <VARDCL> /,B ,ER

<TYPDES>      →   <TYPEXP>/,
              →   <*NAME> -TNAME-

<TYPEXP>      →    BRONLIT
                    ≠PTR≠  :  TPTR,
                    ≠BITS≠ : TBITS,
                    ≠REAL≠ : TREAL,
                    ≠SETLOBJ≠:TSOBJ,
                    ≠MAP≠  : TMAP,
                    ≠ENTRY≠ : TENTR /,B

<TPTR>        →   ≠(≠ ≠*≠  <*NAME>, ≠)≠ -TPTR1-  /,OKPTR   ,  ,ER  ,ER
              →   <*NAME>  ≠)≠  -TPTR2-

<OKPTR>       →   -TPTR3-  [ALLRI]
```

```
<TBITS>        →  ≠(≠  <IEXP>  ≠)≠  -TBITS-

<REAL>         →  -TREAL- [ALLRI]

<TSOBJ>        →  -TSOBJ- [ALLRI]

<TMAP>         →  ≠(≠  <IEXP>  ≠,≠  <*NAME>  ≠)≠ -TMAP-

<TENTR>        →  <TYPEDES>  -TENTRY1- /,
               →  -TENTRY2-  [ALLRI]/

<DMS>          →  <ATTRSP>  <CATTRSP *0> -DIMS- /

<DATAS>        →  <*NAME>  ≠(≠  <IEXP>  ≠)≠ -DATA1-  ,,DATAREST/,ER ,
               →  -DATA2- ,,DATAREST
<DATAREST>     →   ≠=≠  <DATAELT>   <CDATAELT*0> -DATA3- ≠:≠ ,,DATAS/

                    ,ER      ,ER                          ,REST
<DATAELT>      →   <IEXP>  ≠(≠  <IEXP>  ≠)≠ -DATA4-  /,B .  ,ER ,ER

               →  -DATAS-  [ALLRI]/,
<CDATAELT>     →   ,  <DATAELT> / ,B  ,ER

<NMS>          →  <NAME> -NAMESET- [OPEN (NAMESET)] ,,REST /,ER

<ACS>          →  <NAME>  <CNAME*0>  -ACCESS-  ,,REST/,ER

<EXPECTS>      →  <VARDCL> <CVARDCL *0> -EXPECT- /,ER

<TYPS>         →   <*NAME>  -TYPE1- : <TDESPART> ,,REST/ ,ER  ,ER  ,ER

<TDESPART>     →  <TYPEXP> -COMP1- /,
               →  <*NAME>:  -INHERIT-  <TYPEREST> -TYPE2- / ,TYPEREST

<TYPEREST>     →  <TDESCP>  <CTDESCP *0> / ,B

<TDESCP>       →   <*NAME>  ≠(≠  <IEXP> ≠)≠ <TYPEXP> -COMPD-  /,ER , ER ,
               →  <TYPEXP>   /,ER

<FORS>         →  <*NAME> [OPEN(FOR)] -FOR1-  ≠IN≠  <EXP>  -FOR2- ,,REST/
               →  ,ER ,ER ,ER

<SIMPS>        →  <SIMPST>  ,,REST /,B

<SIMPST>       →   BRONLIT
                    ≠CALL≠ : CALLS,
                    ≠CONT≠ : CONTS,
                    ≠QUIT≠  : QUITS,
```

```
                 ≠GO≠      : GOTOS,
                 ≠GOBY≠    : GOBYS,
                 ≠DROP≠    : DROPS,
                 ≠IN≠      :  INS,
                 ≠OUT≠     : OUTS,
                 ≠RETURN≠: RETS   /,
            →  <ASSIGN>

<ASSIGN>    →  <FACTOR> ≠=≠ -ASSIGN1-  <EXP> -ASSIGN2- ,,REST/,ER ,ER ,E

<CALLS>     →  <*NAME> ≠(≠ -CALL1-  <ARGS> ≠)≠ -CALL2- ,,REST /,ER , ,ER
            →  -CALL3-  [ALLRI]

<CONTS>     →  -CONT- [ALLRI] ,,REST/

<GOTO>      →  ≠TO≠ <*NAME> -GOTO-  ,,REST /,ER ,ER

<GOBYS>     →  <EXP>     ,,LABLIST / ,   ,ER  ,ER

<LABLIST>   →  -GOBY1- ≠(≠ <*NAME> <CNAME*0> ≠)≠ -GOBY2- ,,REST/,ER ,ER

<ARGS>      →  <ARG> <CARG*0>

<ARG>       →  -ARG1- <EXP> -ARG2- /, B

<CARG>      →  , <ARG> /,B ,ER

<QUITS>     →  -QUIT- [ALLRI]  ,,REST/

<RESTS>     →  -RETURN- [ALLRI]  ,,REST/

<DROPS>     →  <*NAME> -DROP1- ≠(≠ <EXP> ≠)≠ -DROP2- ,,REST/ ,ER , ,ER ,
            →  -DROP3-  [ALLRI] ,,REST

<INS>       →  <EXP> ≠(≠ <EXP> -IN- ,,REST/,ER  ,ER  ,ER

<OUTS>      →  <EXP> ≠,≠ <EXP> -OUT- ,,REST /,ER  ,ER  ,ER

<EXP>       →  ≠NEW≠ ≠(≠ <*NAME> ≠,≠ <EXP> -NEW1- /,   ,ER  ,ER  ,NEW2
            →  [STARTEXP] <EX> [ENDEXP] / ,ER

<NEW2>      →  ≠)≠ -NEW2- /,ER

<IEXP>      →  [STARTEXP] -IEXP1- <EX> -IEXP2- [ENDEDXP] / ,ER

<EX>        →  <UNOP> <TERM> <EXPTAIL*0> /, ,ER
            →  <TERM> <EXPTAIL*0>
```

```
<UNOP>          →   UNPREC
                     ≠-≠ :8:1,          ≠,NOT,≠ :8:1,
                     ≠.N,≠ :8:1,        ≠-≠ :8:8,
                     ≠.NB,≠ :8:2,       ≠,FB,≠ :8:9,
                     ≠.TYPE,≠ :8:3,  ≠,NELT,≠ :8:10,
                     ≠.ARB,≠ :8:4,   ≠,DEC,≠ :8:11,
                     ≠.OCT,≠ :8:5,   ≠,MIN,≠ :8:12,
                     ≠.MAX,≠ :8:6,   ≠,ROT,≠ :8:13,
                     ≠.TOP,≠ :8:7,   ≠,POW,≠ :8:14,
                                     ≠,NPOW,≠ :8:15

<EXPTAIL>       →   <BINOP> <TERM> /, ,B

<BINOP>         →   BINPREC
                     ≠.C.≠ :1:16,     ≠,EQ,≠ :4:21,        ≠-≠ :5:28,
                     ≠.CC,≠ :1:17,    ≠=≠ :4:21,           ≠+≠ :5:29,
                     ≠.O,≠ :2:18,     ≠,NE≠ :4:22,         ≠*≠ :6:30,
                     ≠.OR,≠ :2:18,    ≠¬≠ :4:23,           ≠/≠ :6:31,
                     ≠V≠ :2:18,       ≠,LE,≠ :4:24,        ≠,IN,≠ :7:32,
                     ≠.EX,≠ :2:19,    ≠,LT,≠ :4:25,        ≠,ELMT,≠ :4:33
                     ≠.EXOR,≠ :2:19,≠,GE,≠ :4:26,
                     ≠.A,≠ :3:20,     ≠,GT,≠ :4:27,
                     ≠.AND,≠ :3:20, ≠<≠ :4:25,
                     ≠^≠ :3:20,       ≠>≠ :4:27

<TERM>          →   BRONLIT
                     ≠TRIM≠      : TRIM,
                     ≠SETOF≠     : SET,
                     ≠TUPLOF≠    : TUP,
                     ≠DIMF≠      : DIM,
                     ≠DEF≠       : DEF,
                     ≠NEWAT≠     : NEWAT,
                     ≠.NL,≠      : NL,
                     ≠.NULT,≠    : NULT,
                     ≠.NULC,≠    : NULC,
                     ≠.NULB,≠    : NULB,
                     ≠.OM,≠      : OM,
                     ≠.TRUE,≠    : TRUE,
                     ≠.FALSE,≠   : FALSE,
                     ≠.NIL,≠     : NIL,
                     ≠.CN,≠      :  CN       /.
                →   <CONSTANT> -CONST- /,
                →   <FACTOR>
                →   ≠(≠ <EXP> ≠)≠

<FACTOR>        →   BRONLIT
                     .F.      : FX,
                     .E.      : EX,
                     .S.      : SX,
                     .CH.     : CHX,
                     .SUB.    : SUBX   /,
                →   <ATOM> -ATOM-
```

```
              →   <DEREF *>   -DEREF- <ATOM>

<DEREF>       →   ≠↑≠ / ,B

<ATOM>        →   <*NAME> ≠≤≠ <EXP> <CEXP*0> ≠≥≠ -OFA-
              →            ≠[≠ <EXP> <CEXP *0> ≠]≠ -OFB-
              →            ≠(≠ -INDEX1- <ARG> <CARG*0>-INDEX2- ≠)≠
                               <TAIL> -COMPR1- / , ,ER  ,ER  ,NOTAIL
              →            <TAIL> -COMPR2- /  ,NOTAIL

<TAIL>        →   ≠:≠ ≠:≠  <ATOM> -QUAL- /,  ER,  ER,
              →   ≠↑≠  <ATOM>
              →   <ATOM>

<NOTAIL>      →   [ALLRI]/

<TRIM>        →   ≠(≠ <EXP> ≠,≠ <EXP> ≠)≠ -TRIM- /,ER  ,ER  ,ER  ,ER

<SET>         →   ≠(≠ <EXP> <CEXP*0> ≠)≠ -SET- /,ER  ,ER  ,ER

<TUP>         →   ≠(≠ <EXP> <CEXP*0> ≠)≠ -TUP- / ,ER  ,ER  ,ER

<DIMF>        →   ≠(≠ <EXP> <CEXP*1> ≠)≠ / ,ER  ,ER  ,ER  ,ER

<DEF>         →   <*NAME> ≠(≠ <EXP> ≠)≠ -DEF1- /,ER,  ,ER  ,ER
              →   -DEF2-  [ALLRI]/

<NEWAT>       →   -NEWAT-  [ALLRI>

<NL>          →   -NL-  [ALLRI]/

<NULT>        →   -NULT-  [ALLRI]/

<NULC>        →   -NULC-  [ALLRI]/

<NULB>        →   -NULB-  [ALLRI]/

<OM>          →   -OM-  [ALLRI]/

<TRUE>        →   -TRUE-  [ALLRI]/

<FALSE>       →   -FALSE-  [ALLRI]/

<NIL>         →   -NIL-  [ALLRI]/

<CN>          →    BRONLIT
                    ≠SETLINT≠        : CN1,
                    ≠SETLPTR≠        : CN2,
                    ≠SETLBCOL≠       : CN3,
```

```
                    ≠SETLCHAR≠        : CN4,
                    ≠BITS≠            : CN5,
                    ≠CHARS≠           : CN6,
                    ≠PTR              : CN7/

<CN1>          →  <CEXP> -CN1- / ,ER

<CN2>          →  <CEXP> -CN2- /,ER

<CN3>          →  <CEXP> -CN3- /,ER

<CN4           →  <CEXP> -CN4- /,ER

<CN5>          →  ≠(≠ <IEXP> ≠)≠ ≠,≠ <EXP> -CN5- /,ER  ,ER  ,ER  ,ER  ,ER

<CN6>          →  ≠(≠ <IEXP> ≠)≠ ≠,≠ <EXP> -CN6- / ,ER  ,ER  ,ER  ,ER  ,E

<CN7>          →  <CEXP> -CN7- /,ER

<FX>           →  -FX- <EXT> /,ER

<EX>           →  -EX- <EXT> /,ER

<SX>           →  -SX- <EXT> /,ER

<CHX>          →  -CHEX1- <EXP> -CHEX2- <CEXP> -CHEX3- /,ER  ,ER

<SUBX>         →  -SUBX1- <EXP> -SUBX2- <CEXP> -SUBX3- <CEXP> -SUBX4-

<EXT>          →  -EXT1- <EXP> -EXT2- <CEXP> -EXT3- <CEXP> -EXT4- /,ER  ,ER  ,E

<REST>         →  ≠;≠  ,,STATEME /,ER

<ER>           →  [ER(N)]  ,,SEMLOOP /

<SEMLOOP>      →  <*ANY>  ;  ,,STATEME /

              END
```

SEMANTIC PROCESSING

The semantic processor translates the polish I, which is highly language dependent, into a standard 'polish II' string of lower semantic level. Two types of transformation are performed during this translation:

1. The declaratory statements of MIDL are processed, and various symbol attributes are entered in a symbol table. Some are used only within the semantic processor, while others, such as size, are passed onto the code generator.

2. Generic operations such as plus are mapped into primitive operations such as integer and real addition, and into calls to the run time library. Along with this, type checking is performed.

The semantic processor maintains a stack which contains symbol table pointers for variables and type descriptors for expression values. Items are read one at a time from the polish I string. If an item is terminal it is placed on the stack. Otherwise an appropriate semantic routine is called which initially pops the stack, then emits code, and pushes a result indicator back onto the stack.

MIDL requires that we collect various types of initialization code as declarations are processed, and then later emit this code in single blocks. We therefore write code fragments to four files:

1. 'Polish II' is the main output file. Whenever the string

< * NAME> READ

appears in polish II, the code generator will read a record of input from the named file.

2. 'INIT' contains code executed the first time each routine is entered.

3. 'REC' contains code executed each time a recursive routine is entered.

4. 'MIDLAST' is an initialization routine called by the system, which allocates stack space for various blocks. Occasionally we must reorder smaller pieces of code. We do this by emitting a special space-holder node HEREIS. We can later backspace to a HEREIS and replace it with a section of code

The symbol table, symtab, is created by the parser and passed to the semantic processor and code generator. It contains the following fields,which must be used identically by all phases and languages:

NAME POINTER:	Pointer to a names array
NCHARS:	Number of characters
PARMNO:	Parameter number
SIZE:	Size
DIMS:	Dimension
VBEG:	Pointer to value table for constants
SCOPE:	Namescope
ADDR:	Machine address
MODE:	One of: bits, real, subr, label nameset or special.

In addition its entries all contain fields of roughly 30 bits which may be used differently for each pass and language. For the MIDL semantic processor these fields are:

RECF:	Flags recursive variables
COMPFLAG:	Flags component names
OFFSET:	Offset in stack
TYP:	Pointer to auxilliary table typtab.
STP:	Pointer to auxilliary table structab.

TYPTAB is a hash table for type descriptions consisting of the following fields:

TYPCL:	General type class
SNAM:	Type name for map, PTR and name types
ARGSZ:	Argument size for maps.

PTRDIMS: Flags dimensigned pointers.

STRUCTAB contains a header entry for each user defined type, giving the following information:

NCOMP:	No. of components
LTLSTR:	Flags type all of whose components are of 'LITTLE' types.
ISHDR:	Flags type stored in type-0 block

Plus the following for each component:

COMPOFFS:	Offset in structure
COMPFBP:	Position in word
COMPFL:	Size of component
COMPNAM:	Name (symtab pointer)
COMPTYP:	Type (typtab pointer)

Polish I Grammar for MIDL

Immediately below we give the grammar for the polish I string read in by the semantic processor, together with algorithms for the major semantic routines. The symbol '*' in the grammar represents a repetition count written on the string by the <clause * N> operation of the parser.

Header Statements

```
<STATEMENT → <HEADER> /
           → <LABEL> <UNLAB> /
           → <UNLAB>
<HEADER>   → <* NAME><NAME *2><RWORD> SUBR1 /*SUBR WITH ARGUMENTS */
           → <*NAME><RWORD> SUBR2  /* SUBR WITHOUT ARGUMENTS */
           → <*NAME><NAME *1><RWORD> SUBR3 /* FUNCTION */
<RWORD>    → RECURSIVE /
             [ALLRI]
<LABEL>    → <* NAME>  LABEL
```

Generators for Header Statements

SUBR1

1. Pop successive parameter names and set *parmno* fields of symbol table. Check for duplicate names.
2. Call SUBR2

SUBR2

1. If this nor the first routine then
   A. Make sure previous routine is closed.
   B. Perform necessary I/O.
   C. Reinitialize nameset access table.
2. Pop routine name and make symtab entry.
3. Emit

   < * NAME> SUBR

   Onto polish II.
4. If the routine is recursive, emit standard code to save return address, copy arguments into the stack, etc.
5. Emit

   INIT READ REC READ

   so that auxilliary files will be inserted.

SUBR3

1. Set FNCTSW to yes, indicating that this is a function.
2. Call SUBR1

Recursive

Set global RECURSIVEFLAG to yes, indicating that this is a recursive routine.

Label

This processes both user and compiler-generated labels.

1. The name at the top of the stack is checked for conflicting uses and its mode set to *label*.

2. The string

   < * NAME> LABEL

   is emitted.
3. Relocation of gotos is left to the code generator.

Compound statements

We maintain an auxiliary stack *cosa* of unclosed compound statements. Each *cosa* entry is a tuple <type, testlabel, body label, end label, elseflag>. The function *getlab(o)* returns unique internal labels.

<IFS> → <EXP> IFT < STATEMENT * O> ELSE <STATEMENT * O> END
→ <EXP> IFT < STATEMENT * O> END
→ <EXP> SIMPLEIF1 <SIMPLE STATEMENT> SIMPLEIF2

IFT

Generator for if-then statement

1. Pop type of <EXP> from stack and check that its a bit string.
2. Generate an endlabel and create a cosa entry.
3. Emit

   ENDLABEL IFNOTGO

ELSE

1. Pop <TYPE, ENDLABEL, ELSEFLAG> from cosa check that type is if-then and elseflag is no.
2. Obtain a new endlabel and emit
   NEWENDLABEL goto
3. Push ENDLABEL onto the stack and call label
4. Push <IFTHEN, -, -, NEWENDLABEL, yes> onto *cosa*.

SIMPLIF1

1. Pop the type of <EXP> and check that it is a bit string
2. Generate endlabel
3. Emit

   END LABEL IFNOTGO

4. Push <SIMPLEIF, -, -, ENDLABEL, -> onto *cosa*.

SIMPLIF2

1. Pop <-, -, -, ENDLABEL, -> from *cosa*
2. Push endlabel onto the stack and call LABEL.

<WHILE> → WHILE1 <EXP> WHILE2 <STATEMENT * O> END

WHILE1

1. Generate test_label and end_label; make a -*cosa*- entry.
2. Push test_label onto stack and call label

WHILE2

1. Pop the type of <EXP> and check that it is a bit string.
2. Write

END_LABEL IFNOTGO

into the polish II.

<UNTIL> → UNTIL1 <EXP> UNTIL2

UNTIL1

1. Generate test_label,body_label,and end_label.
2. make a *cosa* entry.
3. Push testlabel onto stack and call LABEL
4. Emit

BODY LABEL GOTO

UNTIL2

1. Check the type of <EXP>
2. Emit

END LABEL IFGO

3. Push bodylabel onto stack and call LABEL.

<DO> → < *NAME> DO1 <EXP> DO2 <EXP> DO3 <BYPART>

<BYPART> → <BYWORD> <EXPR> DO4 /.
→ /* NO BYPART */ DO5

<BYWORD> → DOWN
→ /* ALLRI */

DO(1)

Processing 'DO < *NAME>'

1. Pop DNAM. Check that is declared bits
2. Write it on polish II.

DO(2)

Processing 'DO < * NAME> = <EXP>'

1. Generate bodylabel, endlabel and testlabel; enter in cosa .
2. Check the type of <EXP>
3. Emit ASSIGN
4. Emit BODY LABEL GOTO
5. Push testlabel and call LABEL
6. Emit *DNAM* twice (for 'dnam = dnam + ')
7. Emit HEREIS

DO3

Processing 'to <EXP>'

1. Check type of <EXP>
2. Backspace to HEREIS

DO4

Processing 'Do <* NAME> = <EXP> to <EXP> by <EXP>

1. If downto flag is set, emit INT MINUS otherwise emit INTPLUS
2. Emit ASSIGN (dnam = dnam + bypart)
3. Push bodylabel and call label
4. Emit DNAM (for dnam = limit)
5. Skip to end of polish II.
6. If downto flag is set, emit LT, else emit GT
7. Emit END LABEL IFGO

DO5

(No 'by' part.)

1. Emit symtab index for constant '1'
2. Go to DO4

DOWN

Set downto flag to yes.

<NAMESET> → < *NAME> NAMESET

NAMESET

1. Check that name is unused so far, assign it the next free namescope number and enter it in symtab.
2. Set curscope to the new namescope
3. Make a *cosa* entry
4. Generate a variable which will be the stack base for the nameset. Enter it in 'basetab' which maps namescopes to base variables.

<QUIT> → QUIT

1. Search thru *cosa* for a 'do', 'while','until', or 'for' entry.
2. Obtain END_LABEL from the cosa and emit

END_LABEL GOTO

<CONT> → CONT

1. Search thru *cosa* for a 'do','while','until' or 'for' entry.
2. Emit

testlabel GOTO

<END> → END

END

1. Pop *cosa*.
2. If popped *cosa* entry is a 'do','while','until' or 'for' type then emit

test_label GOTO

3. If the *cosa* type is any of the above or if-then, define the endlabel by pushing it onto the stack and calling LABEL
4. If the *cosa* entry is a nameset type restore

CURSCOPE = LOCALSCOPE.

5. Otherwise, we have reached the end of a routine . Emit

ENDROUT

Declaratory statements

```
<SIZE>    →  <ATTRSP* O> * <STRWORD> SIZE
<ATTRSP>  → < * NAME> <IEXP>
<STRWORD> →   STACK
          →   NOSTACK
          →  [ALLRI]
<IEXP>    →  IEXP1   <EXP>  IEXP2
<REAL>    →  <NAMES  * O>  *  <STRWORD>  REAL
<DCL>     →  <DCLPART * 1> *  <STRWORD>  DCL
<DCLPART> →  < * NAME>  <TYPEDES>
<TYPDES>  →  <TYPEXP>
          →  < * NAME>    TNAM
<TYPEXP>  →  <IEXP>       IBITS
          →  <* NAME>     IPTR1
          →  <* NAME>     TPTR2
          →               TPTR3
          →               TREAL
          →  <TYPEDES>    TENTR
          →  <IEXP>  < * NAME>  TMAP
```

The declaratory routines create entries in typtab and symtab. They fall into two classes: TBITS, TPTR1, etc.hash type descriptors onto typtab and push pointers to them onto the stack. DCL,SIZE and REAL associate type descriptors with variables. We give special treatment to compile time expressions (<IEXP> in the grammar), pushing their values onto the stack instead of emitting them.

Below are two sample declaratory routines :

TPTR2

This processes a qualified, undimensioned pointer.

1. Pop < * NAME> from the stack and check that is a valid, curently accessable type.

2. Hash the type descriptor

   <PTRTYPE, < *NAME>, 0 , F.>

   onto typtab and push a pointer to it onto the stack.

DCL

1. Pop the iteration counter and loop through 2 and 3:
2. Pop a typtab pointer and a symtab pointer. Check that the mode and typ fields of the corresponding symtab entry are undefined. Set the size, mode, recflag, scope and typ fields.
3. If the current scope is not local, copy the symtab entry into GSYM which stores descriptors of global variables.

<TYPE> → < * NAME> TYPE1 <TYPEXP> COMP1
→ < * NAME> TYPE1 < *NAME> INHERIT <TDESP * 1> TYPE2
→ < * NAME> TYPE1 <TDESP * 1> TYPE2

<TDESP> → < * NAME> <IEXP> TYPEXP

We create structab entries for each user defined type and allocate a pointer to its run-time template which is stored in the heap.

TYPE1

1. Pop the type name from the stack and check that it has no conflicting use.
2. Declare a pointer of the same name to point to the template.
3. Create a header entry in *structab*, which will contain the number of components, etc. Set the *stp* field of the structures symtab entry to point to it.

TYPE2

1. Make *structab* entries for individual components, assigning them fields and offsets.

INHERIT

1. Pop a name from the stack and check that it is a valid structure.
2. Copy its components onto *structab* for the current type.

<DIMS> → < ATTRSP> <CATTRSP> * DIMS

DIMS

1. Pop pairs <NAME, DIMS> from the stack. Check that name is a variable in the current scope but not yet dimensioned.
2. Set its *dims* field

<ACCESS> → <NAME * 1> ACCESS

ACCESS

1. Pop repitition factor and loop over 2 and 3
2. Pop a name and check that it is a nameset.
3. Set the appropiate bit in accesstab.

<EXPECT> → <DCLPART * 1> EXPECT

1. Pop each *DCLPART*, which consists of a name and type. Check that the name has no conflecting uses.
2. Enter the type in *symtab* and set mode to 'function'.
3. Copy the *symtab* entry into *gsym*.

<DATA> → <* NAME> <IEXP> DATA1 <DATAELT * 1> DATA3
→ <* NAME> DATA2 <DATAELT * 1> DATA3

<DATAELT> → <IEXP> <IEXP> DATA4
→ <IEXP> DATA5

Data statements are represented indenticly in polish I and polish II. However, we perform the following checks:

1. The variable being initialized must be declared in the current scope and stored staticly.
2. If it is treated as an array, it must be dimensioned, and the initialized elements must not be out of bounds.

Simple statements

<CALL> → < * NAME> CALL1 <NARG * 1> * CALL2 /
< * NAME> CALL3

<ARG> → ARG <EXP> ARG2

CALL1

Before processing the argument list we set the global *argsw* to yes; as a result code will be generated to assign each argument which is an expression to a temporary.

CALL2

This routine processes the argument list.

1. Pop the repitition factor
2. Emit a call to *rsvstk* to obtain stack space for the argument list. This will have the form:

   no. of arguments *rsvstk* scall

3. Iterate over the arguments. Emit code to push a pointer for each dynamically stored argument onto the stack.
4. Iterate again, emitting the name of each statically stored argument and zero for each dynamically stored argument. Finally emit PLIST to mark the names as a parameter list.
5. Call CALL3

CALL3

Generate actual call

1. Pop the routine name from the stack and check that it is a subroutine or function.
2. Emit

   ROUTNAME SCALL

ARG1

This routine is called before processing each argument or array index.

1. If *ARGSW* = no, return (no problem with array indices)
2. Otherwise emit HEREIS. This allows us to back up and assign the argument to a temporary once we know it is an expression.

ARG2

After processing an argument for index:

1. Check *Argsw*. If its value is no, return.
2. Check whether the last node of polish II is a marker. If so, then
   A. Generate a temporary.
   B. Backup to HEREIS and emit the name of the temporary.
   C. Return to the end of the polish string and emit ASSIGN
   D. Push a symtab pointer to the temporary onto the stack.

<GO TO> → < * NAME> GOTO

GOTO

1. Pop the label name from the stack.
2. If the label has not been defined set its symtab mode to label and enter it on a list of undefined labels.
3. Otherwise, if its mode is not 'label' issue a diagnostic.
4 Emit

NAME GOTO

<GOBY> → <EXP> GOBY1 <NAMES * 1> GOBY2

GOBY1

Check that the type of <EXP> is bits.

GOBY2

1. Check each label as in GOTO and write it onto polish II.
2. Copy the repitition count into polish II.
3. Emit GOBY

<RETURN> → RETURN

RETURN

This routine handles the return statement. If the routine is non recursive we simply emit

RETURN

Otherwise we emit code which does the following:

1. Copys all arguments which are declared as bits, real or entry back into static storage.
2. Resets the pointer to the top of the stack.
3. Calls an assembly language routine to perform the actual return.

Expressions

<EXP> → <TERM> <EXPTAIL * O>

<EXPTAIL> → <TERM> * BINOP

<TERM> → <TERM> * UNOP
→ <SPECIAL>
→ <CONSTANT> CONST
→ <EXP>
→ <FACTOR>

BINOP

This routine processes all binary operators.

1. Pop the code of the operator from the stack.
2. Pop the types of the arguments. Check that they have the same type class.
3. Using an auxiliary bit matrix *'optypes'* check that the combination of operator and operands is legal.
4. If the operands are SETL objects go to 7.
5. If the operation to be performed is arithmetic, emit either appropriate REAL or INT operator node. Otherwise, emit the same node as was found in polish I.
6. Compute the result type and push it on the stack and return.

7. Operations on SETL types generate calls to the run time library. For most run time routines, the operands and results are passed thru four global variables known as registers. A series of low level allocation routines is used to assign and free registers. When necessary, results of subexpressions are copied to temporaries, and the names of the temporaries are placed on the stack. The handling of the run time interface is similar to that used in the current compiler except for the lowest level operations which write calls and assignments onto polish II.

UNOP

This routine generates unary operations. Its logic is similar to that of *BINOP*.

CONST

Constants are simply moved to polish II. Their type descriptors are left on the stack.

<FACTOR> → FX <EXT> /
→ EX <EXT> /
→ SX <EXT> /
→ CHEX1 <EXP> CHEX2 <EXP> CHEX3
→ SUBX1 <EXP> SUBX2 <EXP> SUBX3 <EXP> SUBX4
→ <ATOM> ATOM
→ <DEREF * 1> DEREF <ATOM>

<ATOM> → < * NAME> <EXP * 1> * OFA
→ < * NAME> <EXP * 1> * OFB
→ < * NAME> INDEX1 <ARG * 1> INDEX2 <TAIL> COMPR1
→ < * NAME> <TAIL> COMPR2
→ < * NAME> INDEX1 <ARG * 1> * INDEX2
→ < * NAME>

<TAIL> → <ATOM> QUAL
→ <ATOM>

<EXT> → EX1 <EXP> EX2 <EXP> EX3 <EXP> EX4

Factors may appear in either dexter or sinister positions. We always begin by generating code for the dexter form. For sinister assignments the routine ASSIGN1 executes a rather trivial backup to modify the code.

INDEX1

This routine is called before processing the first argument.

1. Check the mode of name on top of stack.
2. If it is a component, set *cexp* = yes, so only that integer expressions will be accepted.
3. If it is a function, set *argsw* = yes.
4. Otherwise its a variable. If its of type *setlobj*, set *setlindex* = yes. Otherwise, if it is undimensioned, issue a diagnostic.

INDEX2

This is the generator for M(I...).

1. If *cexp* is set, we are indexing a component. Check that there is only a single index and return.
2. If *argsw* is set, this is a function call. We treat it as a subroutine call with an extra argument in which the result is returned. This makes it necessary to obtain a temporary, push it onto the stack and call CALL2. We then emit the temporary onto the polish II and push its type onto the stack.
3. If *setlindex* = no, we are indexing an array. Check that there is only one argument and that it is a bit string. Set *indextodo* = yes to indicate that we must perform an index operation later. We delay the index operation since in the source expression

   A B(I)

   we must add I to the stack offset of B and the component offset of A before indexing.

COMPR1 AND COMPR2

These routines process component extractions. We concentrate on COMPR2 which handles simple components.

1. Pop P and COMPONENT_NAME from the stack. If P is a typetab pointer, set TP = P and go to 5.
2. Otherwise set TP to the type of P. If TP is a pointer obtain its value by emitting .F.1, PS, Heap(BASE-OFFSET_OF_P) or in polish form BASE OFFSET_OF_P INT MINUS HEAP PS 1 INDEXFIELD If the global *indextodo* is set then emit INTPLUS to add the index to the value of the pointer. Go to 6
3. If TP is a name type, emit its address. and if indextodo is set emit INTPLUS. Go to 6
4. If TP is any other type emit a diagnostic.
5. TP is the result type of a previous extract. Check that its a pointer.
6. Map TP and COMPONENT_NAME into a structab index. Obtain the components offset, first bit and length.
7. Emit OFFSET INT PLUS HEAP FIRST_BIT LENGTH INDEXFIELD This gives the equivlent of .F. FIRST_BIT, LENGTH, HEAP(ADDR_OF_STRUCTURE + OFFSET)
8. Push the components type onto the stack.

DEREF

Merely increment counter of derefeneces 'derefct'

ATOM

This generator is called after all necessary component extraction on an atom has been completed.

1. Pop *TP* from the stack. If its a *typtab* pointer go to 4
2. We are processing either a simple name or index operation with no component extracts. Check that arrays are not used without an index and vica versa
3. If *indextodo* is set, index expression is on polish II. If name is stored statically, emit

NAME <u>INDEX</u>

otherwise emit

ADDR_OF_NAME <u>INT PLUS</u> HEAP <u>INDEX</u>

4. If *indextodo* is not set, this is a simple variable reference. Check that the variable has been declared. If its type is 'SETLOBJ' leave it on the stack. Otherwise emit its symtab pointer if the variable is stored statically and

ADDR_OF_NAME HEAP <u>INDEX</u>

if its stored in the HEAP. Set *TP* to the variable's type.

5.Perform deferencing repeat 5 - 8 *derefct* times:

6. Check that *TP* is a pointer type.

7. Emit

PTS 1 <u>FIELD</u>

To extract pointer

8. Emit

HEAP <u>INDEX</u>

9. Replace TP with type it points to
10. Push TP

<u>LITTLE TYPE EXTRACTIONS</u>

These operations require recording of their arguments for consistency with *compr1* and *compr2*.

FX, EX, SX

These routines push an integer between 1 and 3 onto the stack, thereby indicating the type of the extraction

EX1

Emit HEREIS so operands can be reordered.

EX2

Backup to HEREIS and insert another HEREIS

EX3 The polish II now looks like

HEREIS LENGTH-EXPRESSION ORIGIN-EXPRESSION

backup to HEREIS and insert the third expression.

EX4 Restore to the end of the polish II string. Pop the stack and emit the appropriate extraction operator.

ASSIGNMENTS

<ASSIGNMENT> → <FACTOR> ASN1 <EXP> ASN2

<ASSIGN> → <FACTOR> ASSIGN1 <EXP> ASSIGN2

ASSIGN1

We have just emitted the dexter form of a factor. We remove the last operation of polish II, which was INDEX, FIELD, etc. and use it to set the global *ASNOP* indicating the assignment type.

1. If the top item on the stack is the name of a variable, this is a simple SETL assignment. Emit the stack address of the variable and set *ASNOP* = *SINDEX*. Push the type SETLOBJ onto the stack. Return.
2. Remove the last node *LN* from polish II.
3. If LN is a name this is a simple assignment. Put it back on polish II and set *ASNOP* = ASSIGN

4. If *LN* is <u>SCALL</u> then <FACTOR> is a SETL 'OF' operator. Remove another node to determine which dexter routine was being called, and set *ASNOP* to the corresponding sinister routine. Emit 'RESULT' (the name of a library register)
5. Otherwise set *ASNOP* from *LN*.

<u>ASSIGN2</u>

1. Pop the types of source and target. Check that they match.
2. If *ASNOP* is a runtime call, then emit

   <u>ASSIGN</u>

   To place the source in the library's 'RESULT' register. Emit

   ASNOP <u>SCALL</u>

   to call the appropriate library routine
3. Otherwise emit ASNOP.

<u>Code Generation</u>

The polish string which reaches the code generator is language independent. There is no way of telling whether it was created by a LITTLE, MIDL or SETL compiler. The operations contained in polish II are similar to those in the LITTLE VOA. Many will be converted to single machine instructions. Some will be expanded into a series of instructions; others will generate offline calls.

If any optimization is to be performed it will be desireable to be able to identify the output of each operation. This is contrary to the spirit of polish notation, which has no concept of temporaries or redundant expressions. Thus it will probably be necessary to convert the string to some other form. In some cases, this will mean using quadruples. For basic block optimizations it is possible to use a modified polish form which contains temporaries for operations.

For example, the sequence

A = B * C + D

normally written

A B C * D + =

could be written

A B C T1 * T1 D T2 + T2 =

where T1 & T2 hold the results for B * C and B * C + D respectively. In any case this nonstandard representation should be confined to the code generator phase where it is invisible to the general user.

P O L I S H   I I   G R A M M A R

THE POLISH II GRAMMAR IS ≠LANGUAGE INDEPENDENT≠. IT WILL BE THE SAME FOR ALL LANGUAGES USING THIS COMPILER SCHEME.

```
CODE>          →  <ROUTINE*1>

ROUTINE>       →  <HEADER> <INSTRUCTION*0> -ENDROUT-

HEADER>        →  <*NAME> -SUBR-            /* HEADER FOR SUBROUTINE
               →  <*NAME> -FNCT-            /* HEADER FOR FUNCTION

INSTRUCTION>   →  <EXP> <EXP> <BINOP>       /* BINARY OPERATORS
               →  <EXP> <UNOP>              /* UNARY OPERATORS
               →  <EXP> -IFGO-              /* CONDITIONAL BRANCHES
               →  <EXP> -IFNOTGO-
               →  <*NAME> -GOTO-            /* UNCONDITIONAL BRANCH
               →  <EXP> <NAMES*1> -GOBY-    /* GOBY
               →  <NAME*1> -PLIST-          /* FORM PARAMETER LIST
               →  <*NAME> -SCALL-           /* SUBROUTINE CALL
               →  <*NAME> -FCALL-           /* FUNCTION CALL
               →  -RETURN-                  /* RETURN (NON-RECURSIVE
               →  <EXP>  -FRETURN-          /* FUNCTION RETURN
               →  <ASSIGNENT>
               →  <*NAME> -LABEL-           /* LABEL DEFINITION
               →  <DATA>                    /* DATA STATEMENT
               →  <IO>                      /* IO LEFT UNDEFINED
               →  <MISC>
               →  <*NAME> -READ-            /* READ INPUT FROM AUXILIARY FIL

BINOP>         →  -INTPLUS-                 /* INTEGER ARITHMETIC OPERATIONS
               →  -INTMINUS-
               →  -INTMULT-
               →  -INTDIV-
               →  -INTGT-                   /* INTEGER RELATIONAL OPERATIONS
               →  -INTLT-
               →  -INTGE-
               →  -INTLE-
               →  -OR-                      /* LOGICAL OPERATIONS
               →  -EXOR-
               →  -AND-
               →  -REALPLUS-                /* REAL ARITHMETIC OPERATORS
```

```
             →  -REALMINUS-
             →  -REALMULT-
             →  -REALDIV-
             →  -REALGT-                  /* REAL REALATIONAL OPERATORS
             →  -REALLT-
             →  -REALGE-
             →  -REALLE-

<UNOP>       →  -NB-
                -FB-
                -NOT-
                -UNMINUS-

<MISC>       →  <EXP1> <*NAME>  -INDEX-        /* NAME(EXP1)

             →  <EXP1><EXP2><EXP3> -FIELD- /* .F. EXP3,EXP2,EXP1

             →  <EXP1><*NAME><EXP2><EXP3> -INDEXFIELD-
                                    /* .F. EXP3,EXP2,NAME(EXP1)       */

             →  <EXP1><EXP2><EXP3> -EFIELD-
                                    /* .E. EXP3,EXP2,EXP1             */

             →  <EXP1><*NAME><EXP2><EXP3> -EFIELDX-
                                    /* .E. EXP3,EXP2,NAME(EXP1)       */

             →  <EXP1><EXP2><EXP3> -SFIELD-
                                    /* .S. EXP3,EXP2,EXP1             */

             →  EXP1><*NAME><EXP2><EXP3> -SFIELDX-
                                    /* .S. EXP3,EXP2,NAME(EXP3)       */


<ASSIGNMENT> →  <*NAME><EXP> -ASSIGN-
                                    /* -SIMPLE- ASSIGNMENT

             →  <EXP1><*NAME><EXP2> -SINDEX-  /* NAME(EXP1) =EXP2

             →  <EXP1><EXP2><EXP3><EXP4> -SFIELD-
                                    /* .F. EXP3,EXP2,EXP1 = -EXP4-

             →  <EXP1><EXP2><EXP3><EXP4> -SEFIELD-
                                    /* .E. EXP3,EXP2,EXP1 = -EXP4-

             →  <EXP1><EXP2><EXP3><EXP> -SSFIELD-
                                    /* .S. EXP3,EXP2,EXP1 = -EXP4-

             →  <EXP1><*NAME><EXP2><EXP3><EXP4> -SINDEXFIELD-
```

```
                             /* ,F,EXP3,EXP2,NAME(EXP1) = EXP4*/

    →  <EXP1><*NAME><EXP2><EXP3><EXP4> -SEINDEXFIELD-
                             /* ,E,EXP3,EXP2,NAME(EXP1) = EXP4*/

    →  <EXP1><*NAME><EXP2><EXP3><EXP4> -SSINDEXFIELD-
                             /* ,S,EXP3,EXP2,NAME(EXP1) = EXP4*/

   END
```

Applications to other languages

The scheme we have outlined is applicable to a wide variety of languages. In particular, LITTLE is a proper subset of MIDL and can therefore use a subset of the grammars we have given.

The SETL translator has many special problems. In particular, it must perform substantial semantic processing, pass a set of quadruples to the optimizer and finally convert the quadruples to machine code. The current version uses BALM to perform semantic processing and LITTLE generate code. At no point does it produce the quadruples required by the optimizer. We could adapt our MIDL scheme to SETL translation as follows: Use the scanner parser and code generator off the shelf. Write a new semantic phase which will output quadruples rather than polish II and an extra routine at the end of the optimizer to convert the reordered tuples to polish. This scheme is somewhat ambitious, however the initial implimentation could easily convert the tuples to macroized LITTLE. Its greatest advantage would be eliminating the use of BALM.