SETL Newsletter Number 50 September 10, 1971

Three-Phase Parsing Scheme for SETL Kurt Maly

We will give a more detailed specification of the parser of SETL outlined in Newsletter 47. Instead of having two phases as therein mentioned, the parsing process is broken up into three phases. In the current implementation a few restrictions have been placed on the language. The "then block1 if cond1 else...:" construct and its variants will not be available, neither will be the "while iteration" in a compound operator nor the "composite node" and "multiple choice" features in an iff statement.

Phase 1.

The main routine is called lex which in turn calls nt. Nt simply returns the next lexical token from the input string. lex has to condense the input, consisting of a string of characters, into a string of tokens, modify some and insert additional tokens where required. The main data structures are:

i - lexl ... contains the final token string.

ii - tstack.. supplementary stack used for storing body of inverted subroutine-, function- and macro-definitions.

iii - iterbeg. stack for holding the starting tokens of statements.

lex functions are:

1. Collect all macro-definitions and store the body of each macro together with its arguments as a function of its name in the set "mac".

Since macro-definition may appear anywhere in a SETL program (i.e., before or after invocation) we have to collect first all definitions before they can be expanded. Therefore, no macro-definition may appear within a macro-definition.

2. Reverse inverted subroutine-, function- and macro-definitions.

3. Whenever within an inverted function definition a subsequent call is indicated, the reversed function-definition is to be saved and only one call placed into lexl. Only when the next ';' is encountered is the saved definition to be placed thereafter into lexl.

4. Place block markers (either ⟨**lpar**, lpar⟩ or ⟨rpar, rpar⟩ after 'then', before and after 'else', after 'doing', before closing parentheses of 'while head' when doing option was used, after

i - 'for all' iteration header

ii - 'while' iteration header

iii - '(at label)' iteration header

iv - 'initially' iteration header

v - '(load)'

vi - '(store name)',

between two consecutive semicolons (parentheses not counted) where an additional) and ⟩ are placed if no 'end' token is there. The constructs mentioned under i, ii, iii, v, and vi should be preceded by ⟨op, **forl**⟩, ⟨op, whl⟩, ⟨op, **atl**⟩, ⟨op, lod⟩, ⟨op, str⟩ respectively.

5. Replace left parenthesis immediately following a 'lpar' with ⟨**lpar**, lpar⟩ and right parenthesis preceding a 'rpar' with ⟨rpar, rpar⟩.

6. Collect the number of arguments of user-defined operators, functions and subroutines as function of their names in the sets monop, diop, fns, nils respectively.

7. Check for correct ending of compound statements (e.g., 'end if x', 'end while x ∈ ').

8. Replace semicolon ending iff header with ⟨header, : ⟩ and commas after an action node with ⟨head, , ⟩.

Phase 2.

In phase 2 the routine *control* continuously invokes *preparser1* and *postparser1* until an end of file is encountered and places the resulting treetops in lex2. As already mentioned, only the precedence table for *preparser1* and the grammar1 have to be provided for those two routines. *Preparser1* calls the routine *nextoken1* which uses lex1 from phase 1 and a supplementary stack 'unstack' for macro expansion. The functions of *nextoken1* are as follows:

1. Expand macros using unstack.
2. When the token is ∈{end,;,if,then,else,while,when,doing, iff,?,⟨head,,⟩,fal,whl,all,initially,lod,str} or ∈{lpar,rpar} (in which case it is replaced by (or)), two actions are possible. If we are at the beginning of a string to be condensed, place the token in lex2 and go to the start of *nextoken1*. In the other case (i.e., at the end of a condensable string) leave one space free in lex2 for the tree and place the token in the next space of lex2, set the 'begin of condensable string switch' to *true* and return ⟨er,er⟩.
3. Else return the next token from lex1.

Phase 3.

The routine *control* now invokes *preparser2* and *postparser2*, and returns the treetops produced by the postparser calls. For *postparser2* just the grammar2 is needed whereas *preparser2* needs some additional specifications. *Nextoken2* on which it calls is in this case very simple; namely, it returns the next token from lex2. *Preparser2* has to be provided with, in addition to the precedence tables, a usercode block to handle the header of an iff statement. Specifically, when an 'iff' is encountered, set iffbeg to current stackpointer. The tokens '?' and ⟨head,,⟩ will not be condensed until ⟨header,;⟩ is encountered; then a special algorithm condenses the items on the stack, starting at iffbeg, into a binary tree.