July 27, 1971
Kurt Maly

OUTLINE FOR A PARSING SCHEME FOR SETL

The block structure of SETL makes it advisable to parse a program in two phases. In the first phase all simple statements — statements not containing statement blocks (e.g. assignment statements, goto statements, etc.) — and expressions are condensed by preparser 1 and verified by postparser 1. The output of this phase will be a token stream used by the second phase as a lexical token stream (called lex2).

The tokens will be either of the form

a) (type treetop) where 'treetop' is the root of some tree representing either a simple statement or an expression and 'type' indicates which of those two it is.

b) (type token) where 'token' is either a delimiter, a keyword, or any single object which preparser 1 directly emitted to lex2 and 'type' indicates what it actually is.

The second phase takes these tokens as input to preparser 2 and produces a program tree where the leaves, in most cases, are roots of already verified trees. This program tree is then verified in postparser 2.

It might be pointed out that this general scheme has two big advantages, namely

a) a clarity of structure in the program tree and in the grammars,
b) easiness in generating code and it alleviates some implementation problems (such as storage requirements).

Naturally, those two parsers need two different sets of tables. For parser 1 we need:

a) the lexical specification of SETL (used in 'nextoken').
b) the precedence tables to condense simple statements and expressions (used in 'preparser 1').
c) a postparse grammar for those (used in 'postparse 2').

For parser 2 we need:

a) no need for routine 'nextoken' since tokens are produced by parser 1.
b) the precedence tables to condense statements and composite statements to a program tree (used in preparser 2).
c) A postparse grammar for those used in postparser 2. Remember simple statements and expressions are now names of lexical type 'statement' or 'expression'.)

A control program continuously invokes preparser 1, postparser 1, and constructs lex2 till a E-O-F is encountered. Whereafter preparser 2, which uses lex2, and postparser 2 is invoked to execute phase two.

The following modifications in the existing parsing programs are deemed necessary in order to ensure efficient and correct parsing:

For parser 1:

lexical phase:

1) ';', certain keywords (such as 'if', 'then', 'while', 'doing', etc.) and certain delimiters (such as '(' in '(while...', '[' in '[;...', etc.) are immediately sent to lex2.

2) some of them produce more than one token
e.g. then → then (
else →) else (
;; → ;) end ;
(→ indicates what actually is put into lex2.)

3) the scope controllers (e.g. 'end if x') will be emitted as 'end' with (if x) as corresponding datum.

4) 'til name;' causes '(' to be emitted to lex2, skipping it in preparser 1 and when the label 'name' is encounter)end ; are emitted to lex2.

5) upon encountering a function definition with a corresponding call in an expression, scan along till end of it and save definition, call and an associated newly generated variable name in a set (fundefset). Return to preparser this variable name.

6) after 'iff' emit to lex2 name tokens preceded by tokens indicating whether name is the name of a test-node or action-or label-node. If nodes are parenthesized emitting name tokens to lex2 is replaced by returning a series of tokens to preparser 1.

preparser 1: no changes.

postparser 1: no changes.

For the control program:

1) before anything is inserted in lex2 it has to be checked whether it is the beginning of a parenthesized header (e.g. '(while...', '(at...', '(∀x...' etc.); if so it should be preceded in lex2 by a token stating the type of the header (e.g.'whiliter (whil...')

2) all inverted statements (such as: then...if...but; [;...define...]; etc.) have to be reverted to normal form.

3) the saved function definitions have to be parsed by parser 1 and placed in lex2; function call trees have to be produced and together with their associated variable names saved for code generation.

4) construct binary trees for all iff headers and replace all corresponding tokens in lex2 with trees.

For parser 2:

lexical phase: not existing.

preparser 2: accept lexical tokens from lex2 instead of from 'nextoken'.

postparser 2: none.