Assignments for Compiler Construction
Spring 2020

Assignment 2

Please submit solutions on Blackboard by Monday, 17.02.2020 23:59h

2.1 Theory Assignment: Top-down parsing

LL(1) form
Rewrite the following grammar into LL(1) form, by left factoring and eliminate left recursion
The following grammar has been given:
S $\rightarrow$ a B T | a B T w K
B $\rightarrow$ b
T $\rightarrow$ t | ε
K $\rightarrow$ K s | s

a. Modify it so that it becomes suitable for LL(1) parsing (Just remove left recursion).

b. Tabulate the FIRST and FOLLOW sets for all nonterminals of the modified grammar, including which n onterminals are nullable (*i.e.* can derive the empty string).

c. Construct the LL(1) parsing table of the modified grammar if it is suitable for LL(1) parsing, else show why this is not suitable for LL(1) parsing.

2.2 Practical Assignment: VSL Specification

The directory in the code archive ps2_skeleton.zip begins a compiler for a slightly modified 64-bit version of VSL (“Very Simple Language”), defined by Bennett (Introduction to Compiling Techniques, McGraw-Hill, 1990).
Its lexical structure is defined as follows:

- Whitespace consists of the characters ’\t’, ’\n’, ’\r’, ’\v’ and ’ ’. It is ignored after lexical analysis.
- Comments begin with the sequence ’//’, and last until the next ’\n’ char- acter. They are ignored after lexical analysis.
- Reserved words are as follows:
  - **def** - Function definition
  - **begin** - function beginning
  - **end** - end of function
  - **return** - exit from function
  - **print** - Print to console
  - **if then else** - keywords for conditions
  - **while do** - after the while keyword the condition to be evaluated is written and then the do keyword
  - **var** - for variables
- Basic operators are assignment (:=), the basic arithmetic operators ’+’, ’-’, ’*’, ’/’, and relational operators ’=’, ’<’, ’>’. In addition are the following bitwise operators : ’»’ (rightshift), ’«’ (leftshift), ’$\sim$’ (NOT), ’&’ (AND), ’ ^’ (XOR) and ’|’ (OR).

- Numbers are sequences of one or more decimal digits (’0’ through ’9’).
- Strings are sequences of arbitrary characters other than ’\n’, enclosed in double quote characters ’”’.
- Identifiers are sequences of at least one letter followed by an arbitrary se- quence of letters and digits. Letters are the upper- and lower-case English alphabet (’A’ through ’Z’ and ’a’ through ’z’), as well as underscore (’_’). Digits are the decimal digits, as above.

The syntactic structure is given in the context-free grammar on the last page of this document.
Building the program supplied in the archive ps2_skeleton.zip combines the contents of the src/ subdirectory into a binary src/vslc which reads standard input, and produces a parse tree.
The structure in the vslc directory will be similar throughout subsequent problem sets, as the compiler takes shape. See the notes set from the PS2 recitation for an explanation of its construction, and notes on writing Lex/Yacc specifications.

2.2.1 Scanner

Complete the Lex scanner specification in src/scanner.l, so that it properly tokenizes VSL programs.

2.2.2 Tree construction

A node_t structure is defined in include/ir.h. Complete the auxiliary functions node_init, and node_finalize so that they can initialize/free node_t-sized memory areas passed to them by their first argument. The function destroy_subtree should recursively remove the subtree below a given node, while node_finalize should only remove the memory associated with a single node.

2.2.3 Parser

Complete the Yacc parser specification to include the VSL grammar, with se- mantic actions to construct the program’s parse tree using the functions im- plemented above. The top-level production should assign the root node to the globally accessible node_t pointer ’root’ (declared in src/vslc.c).

$program \rightarrow global\_list$
$global\_list \rightarrow global \mid global\_list \; global$
$global \rightarrow function \mid declaration$
$statement\_list \rightarrow statement \mid statement\_list \; statement$
$print\_list \rightarrow print\_item \mid print\_list \; ',' \; print\_item$
$expression\_list \rightarrow expression \mid expression\_list \; ',' \; expression$
$variable\_list \rightarrow identifier \mid variable\_list \; ',' \; identifier$
$argument\_list \rightarrow expression\_list \mid \varepsilon$
$parameter\_list \rightarrow variable\_list \mid \varepsilon$
$declaration\_list \rightarrow declaration \mid declaration\_list \; declaration$
$function \rightarrow$ **def** $identifier \; '(' \; parameter\_list \; ')' \; statement$
$statement \rightarrow assignment\_statement \mid return\_statement$
$statement \rightarrow print\_statement \mid if\_statement$
$statement \rightarrow while\_statement \mid null\_statement \mid block$
$block \rightarrow$ **begin** $declaration\_list \; statement\_list$ **end**
$block \rightarrow$ **begin** $statement\_list$ **end**
$assignment\_statement \rightarrow identifier \; ':' \; '=' \; expression$
$return\_statement \rightarrow$ **return** $expression$
$print\_statement \rightarrow$ **print** $print\_list$
$null\_statement \rightarrow$ **continue**
$if\_statement \rightarrow$ **if** $relation$ **then** $statement$
$if\_statement \rightarrow$ **if** $relation$ **then** $statement \; ELSE \; statement$
$whilestatement \rightarrow$ **while** $relation$ **do** $statement$
$relation \rightarrow expression \; '=' \; expression$
$relation \rightarrow expression \; '<' \; expression$
$relation \rightarrow expression \; '>' \; expression$
$expression \rightarrow expression \; '|' \; expression$
$expression \rightarrow expression \; '\wedge' \; expression$
$expression \rightarrow expression \; '\&' \; expression$
$expression \rightarrow expression \; '>>' \; expression$
$expression \rightarrow expression \; '<<' \; expression$
$expression \rightarrow expression \; '+' \; expression$
$expression \rightarrow expression \; '-' \; expression$
$expression \rightarrow expression \; '*' \; expression$
$expression \rightarrow expression \; '/' \; expression$
$expression \rightarrow \; '-' \; expression$
$expression \rightarrow \; '\sim' \; expression$
$expression \rightarrow \; '(' \; expression \; ')'$
$expression \rightarrow \; number \mid identifier \mid identifier \; '(' \; argument\_list \; ')'$
$declaration \rightarrow$ **var** $variable\_list$
$printitem \rightarrow expression \mid string$
$identifier \rightarrow IDENTIFIER$
$number \rightarrow \; NUMBER$
$string \rightarrow \; STRING$