TDT4205 Problem Set 2
Spring 2012

Answers are to be submitted to *itslearning*, by Wednesday, Feb. 15^{th}, 20:00.

All submitted source code MUST be able to compile and run on asti. Grading is **pass/fail**. Please read the assignment guidelines on itslearning before starting to work on the assignment. Requests for clarifications can be posted on the itslearning forum.

What to turn in
When turning in assignments, please turn in two files:

- (your_username)_answers.pdf : Answers to non-programming questions (Part 1)
- (your_username)_code.zip,tar.gz,tgz : All your code for this assignment, including makefiles and other necessary code (Part 2)

PART 1 - Theory

Task 1.1

A subset of Pascal expressions (ref. http://www.seas.gwu.edu/ hchoi/teaching/cs160d/pascal.pdf) can be abstracted into the grammar

$$
\begin{aligned}
&\mathrm{E} \rightarrow \mathrm{S} \mid \mathrm{S\ r\ S}\\
&\mathrm{S} \rightarrow \mathrm{T} \mid \mathrm{I\ T} \mid \mathrm{S\ a\ T}\\
&\mathrm{T} \rightarrow \mathrm{F} \mid \mathrm{T\ m\ F}\\
&\mathrm{F} \rightarrow \mathrm{i} \mid \mathrm{n}\\
&\mathrm{I} \rightarrow \mathrm{p} \mid \mathrm{m}
\end{aligned}
$$

Tabulate the FIRST and FOLLOW sets for each nonterminal in this grammar in a table like Tab. 1.

Table 1: FIRST and FOLLOW sets for nonterminals

NT	FIRST(NT)	FOLLOW(NT)
E		
S		
T		
F		
I		

Task 1.2

1. What makes a bottom-up parser more powerful than a top-down parser?
2. What kind of parser does GNU Bison generate?
3. Why do we even need a compiler generator? Why not just hand-code the compilers, like other programs?

Task 1.3

1. What characterizes a shift-reduce parser? How is parsing done in practice?
2. Some grammars may result in conflicts in shift-reduce parsers. Describe the difference between shift-reduce conflicts and reduce-reduce conflicts. Give a small example of both conflicts.

PART 2 - Programming: Lexer and parser

The directory in the code archive `ps2_skeleton.tgz` begins a compiler for a slightly modified version of VSL ("Very Simple Language"), defined by Bennett[1].

The lexical structure of VSL is defined as follows:

- *Whitespace* consists of the characters `'\t', '\n'` and `' '`. It is ignored after lexical analysis.
- *Comments* begin with the sequence `'//'`, and last until the next `'\n'` character. They are ignored after lexical analysis.
- *Reserved words* are `FUNC`, `PRINT`, `RETURN`, `CONTINUE`, `IF`, `THEN`, `ELSE`, `FI`, `WHILE`, `DO`, `DONE` and `VAR`.
- *Operators* are assignment (`:=`), and the basic arithmetic operators `'+'`, `'-'`, `'*'`, `'/'` and `'**'` (where the last one represents the power of-operator).
- *Numbers* are sequences of one or more decimal digits (`'0'` through `'9'`).
- *Strings* are sequences of arbitrary characters (except `'\n'`), enclosed in double quote characters (`'"'`). It is an error to break a string across multiple lines.
- *Identifiers* are sequences of at least one letter followed by an arbitrary sequence of letters and digits. Letters are defined as 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 in Fig. 1.

Building the program combines `src/vslc.c`, `src/scanner.l` and `src/parser.y` into a binary called `bin/vslc`, which runs the scanner/parser pair on input from `stdin` (or optionally, a file). The scanner can dump a text representation of tokens and lexemes on `stderr` as it finds them, and the parser can produce a similar representation of syntax trees.

In the subdirectory `vsl_programs` is an example program, along with files of tokens/parse tree generated by a correct scanner/parser pair.

The structure in the `vslc` directory will be similar throughout subsequent problem sets, as the compiler takes shape. See the slide set from the 3^{rd} recitation for an explanation of its construction, and notes on writing Lex/Yacc specifications.

program → function_list
function_list → function | function_list function
statement_list → statement | statement_list statement
print_list → print_item | print_list ‘, ’ print_item
expression_list → expression | expression_list ‘, ’ expression
variable_list → variable | indexed_variable | variable_list ‘, ’ variable |
variable_list ‘, ’ indexed_variable
argument_list → expression_list | ϵ
parameter_list → variable_list | ϵ
declaration_list → declaration_list declaration | ϵ
function → FUNC variable ‘(’ parameter_list ‘)’ statement

statement → assignment_statement | return_statement | print_statement | null_statement |
if_statement | while_statement | block

block → ‘{’ declaration_list statement_list ‘}’
assignment_statement → variable ASSIGN expression |
variable ‘[’ expression ‘]’ ASSIGN expression
return_statement → RETURN expression
print_statement → PRINT print_list
null_statement → CONTINUE

if_statement → IF expression THEN statement FI |
IF expression THEN statement ELSE statement FI

while_statement → WHILE expression DO statement DONE

expression → expression ‘+’ expression | expression ‘-’ expression | expression ‘*’ expression |
expression ‘/’ expression | ‘-’ expression | expression POWER expression |
‘(’ expression ‘)’ | integer | variable | variable ‘(’ argument_list ‘)’ | variable ‘[’ expression ‘]’

declaration → VAR variable_list
variable → IDENTIFIER
indexed_variable → variable ‘[’ integer ‘]’
integer → NUMBER
print_item → expression | text
text → STRING

Figure 1: Context Free Grammar of VSL

Task 2.1

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

Task 2.2

A `node_t` structure is defined in `src/tree.h`. Complete the auxiliary functions `node_init` and `node_finalize` in `src/tree.c`, 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 root node, while `node_finalize` should only remove the memory associated with a single node.

Task 2.3

Complete `src/parser.y` to include the VSL grammar, with semantic actions to construct the program's parse tree using the functions defined in Task 3. The top-level production should assign the root node to the globally accessible `node_t` pointer 'root' (declared in `src/parser.y`).

Task 2.4

The power of-operator (**) is supposed to be right-associative, and with a higher precedence than multiplication, but lower than unary minus. Set the associativity and operator precedence for ** in parser.y.

References

[1] Bennett, J. P.: *Introduction to Compiling Techniques*, McGraw-Hill, 1990