Problem set 2, Lexical analysis and parsing

TDT4205, Spring 2013

Deadline: 20.02.2013 at 20.00 Contact course staff if you cannot meet the deadline.

Evaluation: Pass/Fail

Delivery: Use It's Learning. Deliver exactly two files:

- *yourusername_ps2.pdf*, with answers to the theory questions
- *yourusername_code_ps2.{zip |tar.gz |tar}* containing your modified versions of the files:
  - *scanner.l*
  - *parser.y*
  - *tree.c*

If you submit any other files, they will be ignored.

General notes: All problem sets are to be done **INDIVIDUALLY**. Code must compile and run on `asti.idi.ntnu.no`, or other machines specified by course staff. You should only make changes to the files indicated. Do not add additional files or thrid party code/libraries.

Part 1, Theory

Problem 1, Regular languages

a) Convert the NFA in Figure 1 to a equivalent DFA.

b) Give an example of a language which cannot be described by a regular expression. Briefly explain why the language cannot be recognized.

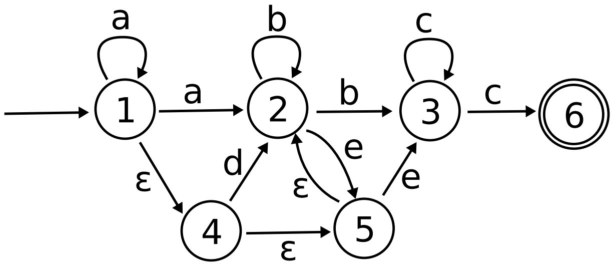

Figure 1: NFA

Problem 2, Grammars

Consider the context-free grammar:

S ::= S S + |S S * |a

a) What is an ambiguous grammar? Why is ambiguity a problem for parsers?

b) Is the grammar above ambiguous or not? Justify your answer.

c) What is an left recursive grammar? Why is left recursive grammars a problem for some parsers?

d) Eliminate left recursion from the grammar.

Problem 3, Parsing

Consider the the context free grammar, representing a simplified version of for and while loops in VSL:

S	::=	F |W |s
F	::=	fAtEdSdd
W	::=	wEdSdd
E	::=	ee |ff |gg
A	::=	aa |bb

a) Tabulate the FIRST and FOLLOW sets for the grammar.

b) Construct the predictive parsing table for the grammar.

Problem 5, Parsing

a) What is the difference between a top-down parser and a botton-up parser?

b) What is the difference between a LL parser and LR parser?

Part 2, VSL

Starting with this assignment, we will create a working compiler for a programing language called VSL (Very Simple Language)[1].

In this assignment, we will implement the lexical analyser, as well as the parser of our compiler. The a specification of the lexical structure and the grammar of VSL can be found below.

Problem 1

The provided archive contains the file `src/scanner.l` which contains parts of a `flex` scanner specification for VSL. Complete this file, based on the VSL specification bellow, so that it properly tokenizes VSL programs.

Problem 2

The structure `node_t` is defined in `src/tree.h`, and will be used for our abstract syntax tree. Complete the following auxiliary functions in `src/tree.h`:

- `node_init` which shuold allocate memory for a node, and initialize it.
- `node_finalize` which should free the memory of a node.
- `destroy_subtreee` which should recursively free the memory for the subtree below a given root node (using `node_finalize` for the nodes).

Problem 3

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

More information about the structure of the provided code and writing flex/bison specifications can be found in the recitation slides.

[1] Based on Bennet, J.P. *Introduction to Compiling Techniques* McGraw-Hill, 1990

VSL Specification

The lexical structure of VSL is defined as follows:

Whitespace consists of the characters '`\t`', '`\n`' and '` `'. It is ignored after lexical analysis.

Comments start with the sequence '`//`', and last until the next '`\n`'.

Keywords are `FUNC, PRINT, RETURN, CONTINUE, IF, THEN, ELSE, FI, WHILE, DO, DONE, FOR, TO` and `VAR`.

Operators are assignment, '`:=`', the basic arithmetic operators '`+`','`-`','`*`','`/`', exponentiation '`**`', and the comparison operators '`>`','`<`','`<=`','`>=`','`==`','`!=`'.

Numbers are sequences of one or more decimal digits, e.g. '`0`' through '`9`'.

Strings are sequences of arbitrary characters (exept '`\n`'), enclosed in double quotes, '`"`'. It's 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 syntatic structure is given in the following grammar. `Courier font` is used for terminals (e.g. tokens). For redability, we represent operators by the lexeme that denotes them, such as `!=` or `:=` as opposed to the token (`ASSIGN`, etc.) returned by the scanner. :

```
Program              ::=  FunctionList
FunctionList         ::=  Function | FunctionList Function
StatementList        ::=  Statement | StatementList Statement
PrintList            ::=  PrintItem | PrintList , PrintItem
ExpressionList       ::=  Expression | ExpressionList , Expression
VariableList         ::=  Variable | VariableList , Variable
DeclarationList      ::=  DeclarationList Declaration | ϵ
ArugmentList         ::=  ExpressionList | ϵ
ParameterList        ::=  VariableList | ϵ
Function             ::=  FUNC Variable ( ParameterList ) Statement

Statement            ::=  AssignmentStatement | ReturnStatement | IfStatement |
                          WhileStatement | ForStatement | NullStatement |
                          BreakStatement | Block

Block                ::=  { DeclarationList StatementList }
AssignmentStatement  ::=  Variable := Expression
ReturnStatement      ::=  RETURN Expression
PrintStatement       ::=  PRINT PrintList
IfStatement          ::=  IF Expression THEN Statement FI |
                          IF Expression THEN Statement ELSE Statement FI
WhileStatement       ::=  WHILE Expression DO Statement DONE
ForStatement         ::=  FOR AssignmentStatement TO expression DO Statement DONE

Expression           ::=  Expression + Expression | Expression - Expression |
                          Expression * Expression | Expression / Expression |
                          Expression < Expression | Expression > Expression |
                          Expression ** Expression | Expression == Expression |
                          Expression != Expression | Expression <= Expression |
                          Expression >= Expression | ( Expression ) |
                          Integer | Variable | Variable ( ArgumentList ) |

Declaration          ::=  VAR VariableList
Variable             ::=  IDENTIFIER
Integer              ::=  NUMBER
PrintItem            ::=  expression | Text
Text                 ::=  STRING
```