Quiz

Compiler Construction, Fall 2012

Wednesday, December 19, 2012

Last Name : _____

First Name : _____

Exercise	**Points**	**Achieved Points**
1	10	
2	10	
3	20	
Total	40	

General notes about this quiz

- This is an open book examination. You are allowed to use any written material. You are not allowed to use the notes of your neighbors.
- You have totally **3 hours 45 minutes**.
- It is advisable to do the questions you know best first.
- Please write the answer of each question on a **separate sheet**.
- Please **return all sheets with quiz questions**, regardless whether you wrote something on them or not. You are free to use your own paper for writing or ask us for paper.

Problem 1: Lexical Analysis (10 points)

Consider the following types of tokens denoting different definitions of identifiers:

OP	$(+ \mid * \mid =)^+$
STRINGID	‘\‘ ‘ $stringLit^+$ ‘\‘ ‘
VARID	$letter\ \ (letter \mid digit)^*\ \ (\ (letter \mid digit \mid \text{‘\_‘})^* \mid\ \text{‘\_‘}(+ \mid * \mid =)^*\ )$

where we take $letter = [a - zA - Z]$, $digit = [0 - 9]$ and $stringLit = [a - zA - Z0 - 9 + * =]$. That is, an identifier can be a string of one or more operator characters (+, *, =), or it can be any string literal between backticks (the identifier name is then taken without the backticks). Or it can start with a letter, followed optionally by a sequence of letters and digits and can then be optionally followed by *either*

- a combination of letters, digits and underscores *or*
- an underscore followed by any number of operator characters

Hence, `x`, `xId_98xyz` and `xId_++=` are valid identifier, but `xId_+xy` is not.
In addition, suppose that the language that we consider has two keywords `case` and `def`, which belong to the token class KEY.

a) **[2 pts]** Determine the tokenizing of the following strings using the longest-match rule, including the type of each token.

i) `big_bob++=‘def‘`

ii) `+*‘case‘type_x==func123_def=+=case**def_77`

b) **[8 pts]** Design a lexical analyzer which can tokenize the above language of operators by giving a deterministic finite automaton. You can use *letter*, *digit*, and *stringLit* where applicable for the respective characters. Please create a simpler version of lexical analyzer that accepts only one token at a time (the longest one possible for a given input), as opposed to one accepting a sequence of tokens; to process a sequence of tokens, the lexical analyzer would be invoked repeatedly until the end of the input is reached.

Problem 2: Grammars (10 points)

Consider the following grammar:

```
Exp -> Exp + Exp
Exp -> Exp < Exp
Exp -> Exp * Exp
Exp -> Exp : Exp
Exp -> num
Exp -> ( Exp )
```

We now want to modify the grammar such that the following precedence rules are enforced:

`*` `+` `<` `:`

where $*$ binds more tightly than any other and : binds less tightly than any other.
In addition, the following associativity should be respected:

- $<$ is non-associative, i.e. an expression like $5 < 6 < 7$ is not permitted
- : is right associative, i.e. $5 : 6 : 7$ should be parsed as $5 : (6 : 7)$
- $+, *$ are left associative, i.e $5 + 6 + 7$ should be parsed as $(5 + 6) + 7$

a) **[8 pts]** Write an unambiguous grammar that derives the same set of strings from Exp as the grammar above and whose parse trees respect the left and right associativity of operators.

b) **[2 pts]** Show the parse tree for $2 : 3 < 4 + 5 : 6 * 7 * 8$ using the unambiguous grammar.

Problem 3: Parsing (20 points)

Consider the following grammar for postfix expressions:

```
E -> EE+
E -> EE*
E -> EE-
E -> num
```

That is, instead of writing $1+2$ we write 12+. Recall that the advantage of this notation is that one does not need parentheses. Precedence is uniquely determined by the order of the operands and operators. For example, instead of writing $3-(4*5)$ in a conventional notation, we write "3 4 5 $*$ $-$", whereas instead of $(3-4)*5$ we write "3 4 $-$ 5 $*$" or "5 3 4 $-$ $*$".

a) **[5 pts]** Use the generalized CYK parsing algorithm to parse the string `2 3 4 + 5 - *`. List the set of all triples (E, i, j) that indicate that E can derive substring from i to j. Give the resulting parse trees.

b) **[15 pts]** Is this grammar ambiguous or not? If it is ambiguous, show an input for which the CYK parser discovers two different parse trees. If it is not ambiguous, prove that every input has at most one parse tree.