Free University of Bozen-Bolzano

Faculty of Computer Science

Bachelor in Applied Computer Science

Dr. Alessandro Artale

Formal Languages and Compilers – A.Y. 2012/2013 – 25.Sept.2013

`Compiler Part`

`Time: 1`^{h}`45 minutes`

STUDENT NAME:
STUDENT NUMBER:
STUDENT SIGNATURE:

`This is a closed book exam. The use of Pencils is not allowed! Write clearly, in the sense of logic, language and legibility. The clarity of your explanations affects your grade. Write your name and ID on every solution sheet.`

1 Exercise: Lexical Analyser [8 Points]

1. **Attribute for a Token.** Describe this notion, what kind of information is associated to it and during what phase of the compilation we need this information.
   Make an example where you show what is a *lexeme* what is a *token* and what is the *attribute*. [2 Point]

2. During the lexical analysis there are 2 kinds of conflicts:

   Case 1. The same Lexeme is recognized by two different RE's.

   Case 2. A given RE can recognize portion of a Lexeme.

   The first case is solved considering the RE listed first. The second case is solved with 2 different techniques: $t1$. Using an Automaton with Lookahead; $t2$. Changing the response to non-acceptance. Describe both techniques $t1$ and $t2$. [4 Points]

3. Assume a Lexer that recognizes just 2 regular expressions, say identifiers and integer numbers. Describe the architecture of the Lexer showing the automaton that will be used to build such a Lexer. [2 Points]

2 Exercise. Top-Down Parsing [8 Points]

Given the following grammar with terminals $VT = \{[,], a, b, c, +, -\}$:

$$\begin{array}{lcl} \texttt{S} & \rightarrow & [\ \texttt{S}\ \texttt{X}\ ] \mid \texttt{a} \\ \texttt{X} & \rightarrow & +\ \texttt{S}\ \texttt{Y} \mid \texttt{Y}\ \texttt{b} \mid \epsilon \\ \texttt{Y} & \rightarrow & -\ \texttt{S}\ \texttt{X}\ \texttt{c} \mid \epsilon \end{array}$$

1. Show the value of the function `FIRST` for all the non terminal symbols. [1 Point]

2. Show the value of the function `FOLLOW` for all the non terminal symbols. [1 Point]

3. Show the parsing table for the LL(1) Top Down Parser recognizing the grammar. [2 Points]

4. Show the stack and the moves of the LL(1) parser on input: "[ab]". [2 Point]

5. Explain the *Backtracking* problem in Top-down parsing and give an example considering the above Grammar. [2 Points]

3 Exercise: Bottom-Up Parsing [11 Points]

Consider the following grammar:

$$
\begin{array}{rcl}
\texttt{SL} & \rightarrow & \texttt{S ; SL} \mid \texttt{S} \\
\texttt{S} & \rightarrow & \texttt{T VL} \\
\texttt{T} & \rightarrow & \textbf{array } \texttt{Idx} \textbf{ of } \texttt{T} \mid \textbf{int} \\
\texttt{VL} & \rightarrow & \textbf{id } \texttt{, VL} \mid \textbf{id} \\
\texttt{Idx} & \rightarrow & \textbf{num}
\end{array}
$$

Show the following:

1. The canonical SLR collection. [4 Points]
2. The transition diagram describing the automaton which recognizes handles at the top of the stack. [2 Points]
3. The parsing table for the SLR parser. [3 Points]
4. The stack and the moves of the SLR parser on input: array 5 of int x. [2 Points]

4 Exercise: Semantic Analysis [6 Points]

Given the following syntax directed definition:

Production	Semantic Rules
$Prog \rightarrow S$	$S.next := newlabel; Prog.code := S.code \parallel gen(S.next\ ':')$
$S \rightarrow S_1\ ;\ S_2$	$S_1.next := newlabel; S_2.next := S.next;$
	$S.code := S_1.code \parallel gen(S_1.next\ ':') \parallel S_2.code$
$S \rightarrow \mathsf{if}\ Test\ \mathsf{then}\ \{S_1\}$	$Test.true := newlabel;\ Test.false := S.next;\ S_1.next := S.next;$
	$S.code := Test.code \parallel gen(Test.true\ ':') \parallel S_1.code$
$S \rightarrow \mathsf{id} := E$	$S.code := E.code \parallel gen(\mathsf{id}.place\ ':='\ E.place)$
$Test \rightarrow \mathsf{id}_1\ \mathsf{relop}\ \mathsf{id}_2$	$Test.code := gen('\mathsf{if}'\ \mathsf{id}_1.place\ \mathsf{relop}.op\ \mathsf{id}_2.place\ '\mathsf{goto}'\ Test.true) \parallel$
	$gen('\mathsf{goto}'\ Test.false)$
$E \rightarrow E_1 + \mathsf{id}$	$E.place := newtemp;$
	$E.code := E_1.code \parallel gen(E.place\ ':='\ E_1.place\ '+'\ \mathsf{id}.place)$
$E \rightarrow \mathsf{id}$	$E.place := \mathsf{id}.place; E.code :=' \ '$

Where:

- The function *newlabel* generates new symbolic labels.
- The function *newtemp* generates new variables names.
- The function *gen* generates strings such that everything in quotes is generated literally while the rest is evaluated.
- The attribute *code* produces the three-address code.
- The attribute id.*place* represents the name of the variable associated to the token id.
- The attribute relop.*op* represents the comparison operators (i.e., $<, <=, =, <>, >, >=$).
- The symbol $\parallel$ means string concatenation.

Given the input:

```
if y > w then {
                    y := x + z};
x := z + v
```

Show the following:

1. The annotated parse tree (without the *code* attribute) for the input together with the values of the attributes. [2 Points]
2. The three-address code produced by the semantic actions for the given input. [2 Points]
3. Give the definition of *inherited attribute*. For the rule $S \rightarrow \mathsf{if}\ Test\ \mathsf{then}\ \{S_1\}$, show what are the synthesized attributes and what are the inherited attributes (mark with **s** the synthesized and with **h** the inherited attributes in the above semantic rules). [2 Points]