Compiler Design

Lexical analysis: Finite automata, Regular sets and regular expression, lexical analyzer design: interface with input, parser and symbol table, token, lexeme and patterns. Difficulties in lexical analysis. Implementation. Regular definition, Transition diagrams.

Q: Consider the following Lex-like description of two tokens:

abcd // Token 1

(abcd)* e // Token 2

(a) Draw the minimum DFAs corresponding to these regular expressions.

(b) Draw a single minimum DFA that accepts both the regular expressions. Indicate against each final state, the token recognized by that state.

(c) Give an input string whose complete tokenization (extraction of all the tokens in the string) would take time that is **quadratic** in its length. Explain how the tokenization will take place for your input string.

Syntax Analysis: Context-free grammars (CFG), ambiguity, associativity, precedence, top down parsing, recursive descent parsing, transformation on the grammars, predictive parsing, bottom up parsing, LR parsers (SLR, LALR, LR).

Q: Consider the following hierarchy of Grammars:

LALR(1) <= LR(1) <= Unambiguous but not LR(1) <= Ambiguous

For each of the following grammars, determine where does the grammar belong in the above hierarchy. No marks for guessing, give suitable reasons to support your answer. In each case, uppercase letters (A, B, S) denote the nonterminals, lowercase letters (a, b, c, d, e) denote the terminals, and ε denotes the empty string.

(i) $A \rightarrow a\ A\ a \mid \varepsilon$

(ii) $A \rightarrow a\ A\ b \mid A\ A \mid \varepsilon$

(iii) $S \rightarrow A \mid B \mid \varepsilon,\ A \rightarrow aAa \mid S,\ B \rightarrow bBb \mid S$

(iv) $S \rightarrow A\ a \mid C\ A\ c \mid B\ c \mid C\ B\ a,\ A \rightarrow d,\ B \rightarrow d,\ C \rightarrow c\ c$

(v) $S \rightarrow a\ S\ a \mid b\ S\ b \mid \varepsilon$

(vi) $S \rightarrow a\ A\ d \mid b\ B\ d \mid a\ B\ e \mid b\ A\ e,\ A \rightarrow c,\ B \rightarrow c$

Q: Consider the following grammar where nonterminals are E, L, terminals are (, a,), and E is the start symbol.

$E \rightarrow (\ L\ ) \mid a$

$L \rightarrow E\ L \mid E$

a) Construct the SLR parse table for the grammar. Clearly show all the steps in the construction.

b) Show the moves (stack and actions) of an SLR parser for the input

((a) a (aa))

Q: Explain why there cannot be a shift-shift conflict in LR parsers?

Q: Consider the rules for computation of FOLLOW set:

a) Place $ in FOLLOW(S)

b) If there is a production A → α B then everything in FOLLOW(A) is in FOLLOW(B)

c) If there is a production A → α B β then everything in FIRST(β) (except ε) is in FOLLOW(B).

d) If there is a production A → α B β and First(β) contains ε then everything in FOLLOW(A) is in FOLLOW(B)

Your friend suggested that the rule (b) should be changed as follows:

b') If there is a production A → α B then everything in FOLLOW(B) is in FOLLOW(A).

Can you explain with an example why your friend is wrong?

Syntax-Directed Definitions and Translations: inherited and synthesized attributes, dependency graph, evaluation order, bottom up and top down evaluation of attributes, L- and S-attributed definitions, syntax-directed translation schemes.

Q: Consider the following grammar that represents a list of assignment statements:

$$
\begin{array}{rcl}
L & \rightarrow & L\ S \\
 & | & S \\
S & \rightarrow & \text{id} = E \\
E & \rightarrow & E\ +\ E \\
 & | & E\ *\ E \\
 & | & (E) \\
 & | & \text{const} \\
 & | & \text{id}
\end{array}
$$

We want to write a translator to estimate the total execution time of a program accepted by the above grammar. Assume the following costs of execution for primitive operations:

Operation	Description	Cost
store (=)	store value in memory	s
loadV	load a variable's value from memory	lv
loadI	load a constant	li
+	addition	a
*	multiplication	m

a) Add Syntax Directed Definition (SDD) to compute execution cost for input programs. Assume that every read of a variable will require it to be loaded from the memory. Also, mention clearly the type (synthesized/inherited) of every attribute that you use.

b) What is the cost of the following program with your estimator as described in (1a):

x = y + (3 * 5)

z = y + x

x = y * y

c) Assuming that the target machine to run the program has an infinite supply of registers, so that a variable need to be loaded only once (as opposed to once for every read in (a)), what is the cost of program in (b)?

d) Modify the translator in part (a) so that it computes the cost when an identifier is loaded at most once.

e) Are your SDDs (in parts a and d) S-attributed or L-attributed or none?

Q: This grammar generates binary numbers with a decimal point:

$S \rightarrow L.L \mid L$

$L \rightarrow LB \mid B$

$B \rightarrow 0 \mid 1$

Design an L-attributed definition to compute S.val, the decimal number value of the input string. For example, the translation of string **101.101** should be the decimal number 5.625. Draw parse tree for string **101.101** and show how the rules compute the value. DO NOT TRANSFORM THE GRAMMAR.

Q: Assume following grammar for array types:

$T \rightarrow B\ C$

$B \rightarrow \text{int}$

$B \rightarrow \text{float}$

$C \rightarrow [\ \text{num}\ ]\ C$

$C \rightarrow \varepsilon$

a) Write rules to compute attributes type (capturing the type expression), and width (capturing the number of bytes required) of all the grammar symbols. Assume that integer takes 4 bytes and float takes 8 bytes. You may use both inherited and synthesized attributes. However, make sure that the attribute equations are put in the right places so that evaluation order of the attribute equations remains correct.

b) Make parse tree of the string **int[4][5][6]** and show the value of attributes at each of the nodes using the above grammar and the rules developed in part (a).

You will not get credit for this part if the answer to the part (a) is not correct.

Consider the following grammar for statements which includes the C-style *for*-statement:

$$S \rightarrow \texttt{for (}e_1\texttt{; }e_2\texttt{; }e_3\texttt{) } S \mid \texttt{break} \mid \texttt{continue} \mid assignment \mid S\texttt{; } S$$

Assume that the meaning of a *for*-statement is the same as:

```
e1;
while (e2) {
    S;
    e3;
}
```

Write an incremental syntax-directed translation (done along with parsing) to generate three address code for the grammar that uses synthesized attributes only. Assume that e_2 has attributes *truelist* and *falselist*, and *stmt* has the attribute *nextlist*. Also assume that you have the standard functions: *backpatch*, *gen*, *nextinstr*, *makelist* and *merge*. You may use other attributes and functions but state clearly what they do. You may also rewrite the grammar without changing the language generated.

Assignment 3

1:

Given is a grammar for statements that includes a `switch` statement.

$$\begin{array}{lcl} S & \rightarrow & \texttt{switch}\ E\ \{\ clauses\ \} \mid \texttt{break} \mid \texttt{others} \mid S; S \\ clauses & \rightarrow & clause;\ clauses \mid clause \\ clause & \rightarrow & \texttt{case const :}\ S \mid \texttt{default} : S \end{array}$$

We would like to generate 3-address code for switch statements. The nature of the code that we would like to generate is illustrated below. L_{next} is the label of the statement following the switch statement in execution order.

```
switch E {                          code to evaluate E in t
   case V1: others1;                if t <> V1 goto L1
   case V2: others2; break;         code for others1
   default: others3;           L1:  if t <> V2 goto L2
}                                   code for others2
                                    go to Lnext
                               L2:  code for others3
```

2:

Give L-attributed definitions to generate 3-address code for `switch` statements under the following assumptions that you have to follow strictly.

(a) As code is generated, it is dumped into a global array (incremental translation). Except for the code array no other global data should be used.

(b) The symbol `others` stands for statements other than `switch` and `break` and includes statements that alter flow of control. Therefore, to avoid jumps-to-jumps, each statement should be passed an inherited attribute called *next*.

(c) Assume that `others` and E produce 3-address code as required. In addition E produces a synthesized attribute *addr* which has the name of the variable that would hold its value.

Code Generation: Definition, machine model, basic blocks and flow graphs, Simple code generator, peephole optimization.

Text Book:

1. Aho, Alfred; Sethi, Ravi; Ullman, Jeffery; **Compiler Principles, Techniques and Tools.**

   (Or a later edition with Monica Lam as an additional author)