Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue&Thu 12:00-13:15 @SH 212, Lab: Fri 10:00-11:15/11:30-12:45 @EB 103

Lab 4: Top-Down Parsing

Learning Objectives Upon successful completion, students will be able to:

- construct LL(1) parsing tables for context-free grammars,
- trace parsing steps with a given parsing table, and
- manually implement an LL(1) parser

Preparation

You can either download the file `lab4.zip` from D2L to your laptop, or copy the file `/u/li/321/lab4.zip` to your CS Linux account. After unzipping the file, enter the directory `lab4`. This lab consists of two parts, one for paper and pencil problems, and one for programming in Java.

A Quick Review on Top-down Parsing

Top-down parsing is one of the two main parsing strategies (the other is bottom-up parsing). It builds a parse tree from top down. Starting with the root node, it repeatedly predicts the next production to apply, until the whole parse tree for the given input is constructed.

LL parsing is often used as a synonym for top-down parsing (which we do, too). This is based on the fact that in most cases, an input token sequence to a parser is read from left-to-right.

For a grammar to be suitable for top-down parsing, it cannot be ambiguous, nor left-recursive.

The steps involved in building a top-down parser include finding the first and follow sets, the lookahead symbols for productions, and the LL parsing table.

Part 1. Manual Implementation of Top-Down Parsers

Recall that a top-down parser consists of a set of parsing routines, one for each nonterminal. Within a parsing routine, there are a set of clauses, one for each production of the nonterminal. Each clause consists of a sequence of actions, *matches* and *calls*, which correspond to the rhs of the production. The clauses are selected by their lookahead symbols.

In this part, you'll manually implement LL(1) top-down parsers for two small grammars, both from this week's lecture.

1. Grammar1 and its parsing table:

```
0. Program0 -> Program "$"
1. Program  -> "begin" StmtList "end"
2. StmtList -> Stmt ";" StmtList
3. StmtList ->            // epsilon
4. Stmt     -> "simple"
5. Stmt     -> "begin" StmtList "end"
```

	begin	simple	;	end	$
Program	1				
StmtList	2	2		3	
Stmt	5	4			

   The file `Parser1.java` contains partial code for an LL(1) parser for this grammar. Since this grammar's tokens are very simple, the program uses Java's Scanner for lexing.

Complete this parser by providing the code for the parsing routines. Compile and test the program. There is a set of tests for this parser in `test1*.in`.

2. Grammar2 and its parsing table:

```
0. E0 -> E "$"
1. E  -> T E1
2. E1 -> "+" T E1
3. E1 ->              // epsilon
4. T  -> P T1
5. T1 -> "*" P T1
6. T1 ->              // epsilon
7. P  -> id
```

	id	+	*	$
E	1			
E1		2		3
T	4			
T1		6	5	6
P	7			

The token `id` is defined to be sequence of letters. Implement an LL(1) parser for this grammar in `Parser2.java`. A set of test inputs are available in `test2*.in`.

Hint: To check whether a lexeme string is an `id` token, you could use Java String's `matches` method:

```
lexeme = scanner.next();
if (lexeme.matches("[a-z]+"))
  ...
```

Part 2. Paper-and-Pencil Problems

1. (a) Consider the following grammar for prefix expression:

   1. $E \rightarrow +\,E\,E$
   2. $E \rightarrow -\,E\,E$
   3. $E \rightarrow \mathsf{id}$

   Construct an LL(1) parsing table for this grammar. Is this grammar ambiguous?

   (b) Now consider the following grammar for postfix expression:

   1. $E \rightarrow E\,E\,+$
   2. $E \rightarrow E\,E\,-$
   3. $E \rightarrow \mathsf{id}$

   Is this grammar ambiguous? Prove or disprove.

2. Consider the following grammar:

   0. $S_0 \rightarrow S\,\$$ // augmented production
   1. $S \rightarrow AB$
   2. $A \rightarrow aA$
   3. $A \rightarrow \epsilon$
   4. $B \rightarrow bB$
   5. $B \rightarrow b$
   6. $B \rightarrow (\,S\,)$

   (a) Compute the First and Follow sets for the nonterminals.

   (b) Compute the Predict (i.e. Lookahead) sets for the productions.

   (c) Construct an LL(1) parsing table based on these sets.

   (d) Is this grammar LL(1)? Why or why not?

3. Consider the following grammar:

   0. $S \rightarrow E\,\$$ // augmented production
   1. $E \rightarrow \mathsf{id}\,M$
   2. $M \rightarrow = E$
   3. $M \rightarrow N$
   4. $N \rightarrow +\,\mathsf{id}\,N$
   5. $N \rightarrow \epsilon$

   We want to verify that this grammar is LL(1).

   (a) Compute the First sets for the right-hand side of all productions; the Follow sets for all nonterminals; and the Predict sets for all productions.

   (b) Construct an LL(1) parsing table for this grammar.

   (c) Use the table to trace the parsing process of input $\mathsf{id} = \mathsf{id} = \mathsf{id} + \mathsf{id}$.