Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue 10:00-11:50 @SRTC 155, Labs: Tue 14:00-15:50 & Wed 12:00-13:50 @FAB 88-10

Lab 4: Top-down Parsing and JavaCC

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
- convert a grammar to a JavaCC program.

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 sequence to a parser is read in 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.

Exercises

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 LL(1) parsing table:

	a	b	c
S	1	1	1
X	2	3	3
Y		4	5

(a) Perform reverse engineering. Provide a context-free grammar that matches the above parsing table. Make sure you label the productions correctly.

(b) Does your grammar contain recursion? If not, can you come up with a matching grammar that does?

(c) Do your grammars contain ϵ-productions? If not, can you come up with a matching grammar that does?

(d) What can you say about the relationship between an LL(1) parsing table and a corresponding context-free grammar? Does an LL(1) parsing table uniquely defines a context-free grammar? What aspects of a context-free grammar does an LL(1) parsing table capture?

4. 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 $\mathsf{id} = \mathsf{id} = \mathsf{id} + \mathsf{id}$.

Converting Grammar to JavaCC Code

For most part, using JavaCC to implement a top-down parser is straightforward. One just need to specify the grammar in EBNF form using JavaCC's notation.

Let's see how to convert the grammars appeared in this lab into JavaCC code.

Grammar1a

The grammar from Program 1(a) and its JavaCC code is shown below. You can see that the JavaCC's EBNF notation is very close to the original one. Terminals are either quoted (for literal tokens) or are in angle brackets; nonterminals are attached with a pair of parenthesis. Productions are represented in the form of parsing routines. In the current simple form, there is no argument to, nor return value from a parsing routine.

Compile and run the JavaCC program to see how it behaves.

from Grammar1a.jj

```
TOKEN: { <ID: (["A"-"Z"]|["a"-"z"])+> }

// E0 -> E $    /* augmented production */
void E0(): {}
{
  E() <EOF>
}

// E -> + E E | - E E | id
void E(): {}
{
  "+" E() E() | "-" E() E() | <ID>
}
```

Grammar1b

Here is the JavaCC code for the grammar from Problem 1(b). Try to compile it and see what happens.

from Grammar1a.jj

```
TOKEN: { <ID: (["A"-"Z"]|["a"-"z"])+> }

// E0 -> E $    /* augmented production */
void E0(): {}
{
  E() <EOF>
}

// E -> E E + | E E - | id
void E(): {}
{
  E() E() "+" | E() E() "-" | <ID>
}
```

Since JavaCC is a top-down parser generator, it does not allow left-recursion.

Grammar2

JavaCC code for the grammar from Problem 2.

from Grammar1a.jj

```
// S0 -> S $    /* augmented production */
void S0(): {}
{
  S() <EOF>
}

// S -> A B
void S(): {}
{
  A() B()
}

// A -> a A | ϵ
void A(): {}
{
  [ "a" A() ]
}


// B -> b B | b | ( S )
void B(): {}
{
  "b" B() | "b" | "(" S() ")"
}
```

JavaCC does not have a way to specify ϵ-production directly. But it supports the optional operator, `[ ]`, which can be used to indirectly express an ϵ-production.

Try to compile this program. Notice that JavaCC compiler generates a warning, `"Choice conflict ..."`. This is because by default, JavaCC is targeted to generate an LL(1) parser, while this grammar is not LL(1).

Try to further compile the Java program, `Grammar2.java`. What happens?

To fix this problem, we have two options:

1. Ask JavaCC to generate an LL(2) parser. This can either be done with a global declaration at the top of the program:

   ```
   options { LOOKAHEAD=2; }
   ```

   or done locally by inserting `LOOKAHEAD(2)` at the beginning of the first conflicting production.

   ```
   LOOKAHEAD(2) "b" B() | "b" | "(" S() ")"
   ```

2. Use left-factoring technique to fact out the common-prefix `b`, which was the source of the conflict.

   ```
   "b" [B()] | "(" S() ")"
   ```

Try these methods with the JavaCC code and see how they work.

Exercises

Convert the grammar from Problem 4 to JavaCC code.