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 3: Grammar Transformations

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

- identify parsing-related issues in context-free grammars;
- transform a grammar to a form suitable for parsing;
- convert a grammar to Bison and JavaCC specifications.

Preparation

You can either download the file `lab3.zip` from D2L to your laptop, or copy the file `/u/li/321/lab3.zip` to your CS Linux account. After unzipping the file, enter the directory `lab3`. This lab consists of two parts, one for paper and pencil problems, and one for programming with Bison and JavaCC. If you don't have Bison and JavaCC on your laptop, you should use the linuxlab machines for this part.

As before, the problems that are marked with an asterisk ($\star$) are a little more challenging.

A Quick Review on Grammar Transformations

A context-free grammar defines a context-free language. However, the correspondence between grammar and language is not one-to-one. Multiple grammars can define the same language. In such case, the group of grammars are said to be *equivalent* to each other. Equivalent grammars do not necessarily have the same characteristics, *e.g.* some may be ambiguous, some may be left-recursive, while others may be not.

To prepare a grammar for parser construction, it is often necessary to *transform* the grammar to an equivalent, but more parser-friendly grammar. The number one reason for doing this is to eliminate ambiguity in a grammar. For topdown parsers, there are two other reasons: to eliminate left-recursion and to factor out common-prefix. Here is a brief recast of these three concepts.

Ambiguity A grammar is ambiguous if it can derive a sentence with more than one parse tree. Ambiguity in a real-world programming language's grammar can always be eliminated. However, there are *inherently* ambiguous context-free languages, for which no unambiguous context-free grammar exists. Also, in general, "whether a given grammar is ambiguous" is an *undecidable* problem.

Left Recursion A grammar is left-recursive if it has a nonterminal that can derive itself as the first symbol in a sentential form, *i.e.* $E \overset{+}{\Rightarrow} E\alpha$. Left recursions are not always explicit, but in general are easy to detect. Left recursions can always be eliminated.

Common Prefix A common-prefix is a prefix of a sentential sequence that can be derived by multiple productions of the same nonterminal. Like left recursion, common prefixes are not always explicit, and can be hard or sometimes impossible to eliminate.

There are established transformation rules for these cases. However, in general, there are no comprehensive algorithms. Grammars have to be studied case by case to decide whether transformations are needed, and what transformations should be used.

Part 1. Paper-and-Pencil Problems

1. Show that the grammar

   $A \rightarrow -A$
   $A \rightarrow A - \mathsf{id}$
   $A \rightarrow \mathsf{id}$

   is ambiguous by finding a sentence that has two different parse trees.

   Now define two unambiguous grammars for the same language:

   (a) One in which the prefix $-$ op binds stronger than the infix $-$ op.

   (b) One in which the infix $-$ op binds stronger than the prefix $-$ op.

2. Is the following grammar ambiguous? Provide a concrete example or a convincing argument.

   $E \rightarrow E + T$
   $E \rightarrow T$
   $T \rightarrow -E$
   $T \rightarrow \mathsf{id}$

3. Eliminate left-recursion in the following grammars:

   (a) $E \rightarrow E - x$
   $E \rightarrow y$

   (b) $E \rightarrow E + \mathsf{id}$
   $E \rightarrow E - \mathsf{id}$
   $E \rightarrow \mathsf{id}$

4. To eliminate left-recursion, can we simply transform each left-recursive production *individually* to a right-recursive one? *E.g.* can we transform the grammars in the previous problem directly to

   (a) $E \rightarrow x - E$
   $E \rightarrow y$

   (b) $E \rightarrow \mathsf{id} + E$
   $E \rightarrow \mathsf{id} - E$
   $E \rightarrow \mathsf{id}$

5. Which of the following grammars is/are left-recursive? For the one(s) you identify, eliminate the left-recursion:

   (a) $E \rightarrow F\,E \mid a$
   $F \rightarrow b \mid c \mid \epsilon$

   (b) $E \rightarrow \{\,F\,\}E \mid a$
   $F \rightarrow b\,F \mid b$

6. ($\star$) In some languages, assignment of the form $\mathsf{id} = E$ is a valid expression. Consider the following grammar for such expressions:

   $E \rightarrow \mathsf{id}\ = E$
   $E \rightarrow T\,\{+T\,\}$
   $T \rightarrow F\,\{*\,F\,\}$
   $F \rightarrow \mathsf{id} \mid \mathsf{num}$

   Upon seeing an id token, a parser based on this grammar would not be able to decide whether the id is the left-hand side of an assignment or production it is a standalone id in an arithmetic expression. In other words, there is a hidden common-prefix between the first two productions. Can you use the left-factoring technique to fix this problem?

Part 2. Programming Exercises

Parser generators can be used to verify a grammar's properties. For instance, if a grammar compiles without conflicts with Bison, then we know it is not ambiguous; if it also compiles without conflicts with JavaCC, then we further know that it does not have left-recursion or common-prefix (assuming JavaCC is using the default lookahead setting).

Note that the reverse is not as useful. If Bison reports conflicts on a grammar, you may not conclude that the grammar is ambiguous, since unambiguous grammar can also cause conflicts in Bison (*e.g.*, an LR(2) grammar). Similarly, if JavaCC reports conflicts on a grammar, we may not know if it is ambiguity or common prefix that caused the problem. (Left-recursion always gets explicitly reported by JavaCC.)

Exercises

1. In the `lab3` directory, you'll see two program files, `grammar1.y` and `grammar1.jj`. They are the Bison and JavaCC encoding of the grammar from Problem 1. Both can run as standing-alone parsers: `grammar1.y` has a very simple lexer included in it, while JavaCC automatically generates a lexer for every `.jj` program.

   Try to compile these programs:

   ```
   linux> bison grammar1.y
   linux> javacc grammar1.jj
   ```

   What do you see? What can you say about Grammar1 based on these messages?

2. Try to encode all the other grammars of Part 1, including those you come up with, in Bison and JavaCC specifications. Use a consistent naming convention based on problem numbers, *e.g.* `grammar1a.y`, `grammar1b.y`, `grammar2.y`, etc.

   Compile these programs to see if they cause conflicts. For the grammars you come up with, do Bison and JavaCC validate them?