Compilers [Spring 2021]

Programming Assignment III

Objectives

1. To become familiar with bison (a popular, yacc-compatible parser generator).
2. To implement a parser for programs written in DJ.
3. To practice writing context-free grammars (CFGs) by specifying DJ's grammar.

Due Date: Sunday, February 28, 2021 (at 11:59pm).

Assignment Description

This assignment asks you to implement a basic parser for our dj2dism compiler. You will use bison (which Section 5.5 of the textbook describes well) to generate the parser.

First, download this CFG-less *dj.y*: http://www.cse.usf.edu/~ligatti/compilers/21/as3/dj.y
Modify that *dj.y* by declaring appropriate precedence directives in the first section and a correct CFG for DJ in the second section. For this assignment, leave the actions in your CFG empty, except that every rule for the CFG's starting symbol should have the `{return 0;}` action (Assignment IV will ask you to fill in the CFG actions to build an AST).

For full credit on Assignments III and IV, your parser must have *no* conflicts (shift/reduce or reduce/reduce) and *no* right-recursive rules, except that right-recursion is allowed in the category for "expressions" (right-recursion isn't a problem in this category because your precedence directives will guide the parser and ensure that space inefficiencies are automatically avoided when possible). Using left recursion will also make Assignment IV easier to complete.

Hints

You may find it helpful to study the format of the empty-action CFG for DISM posted at: http://www.cse.usf.edu/~ligatti/compilers/21/as3/dism-cfg.y

When you find that your parser has conflicts (like I did on my first attempt at this assignment), please examine your bison output file to find the cause of the conflict and then think calmly and carefully about how to modify the CFG to remove the conflict. Please do not just try to "hack" through the problem! Hacked-up grammars are normally complicated and hard to understand. Part of your grade on this assignment will be determined by the simplicity/elegance of your grammar.

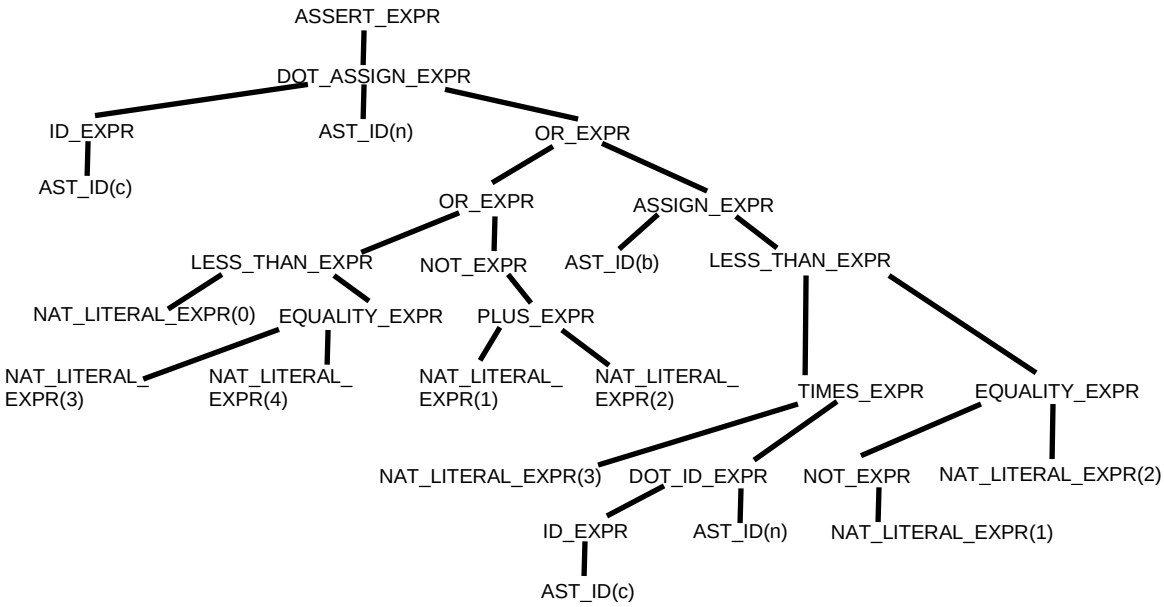
Precedence and Associativity of Operators

A key challenge of this assignment is to declare precedence directives correctly and only in ways that immediately make sense to outside readers of your code.

Consider the following DJ expression (from file *good11.dj*):

```
assert c.n = 0 < 3 == 4 || ! 1 + 2 || b = 3 * c.n < !1 == 2;
```

A correct and complete DJ parser (as yours should be after you complete Assignment IV) would produce the following AST for the *good11.dj* expression above.



A correct declaration of precedence directives (%left, %nonassoc, %right) in this assignment will help produce these groupings. Notice, for instance, that the DOT operator has the highest precedence. The (left-associative) PLUS and MINUS operators should have equal precedences, lower than TIMES. The EQUALITY and LESS operators should be non-associative, while ASSIGN is right associative. Although the NOT and ASSERT operators are unary (i.e., only have one operand), you can declare them to be right associative for the sake of ordering their precedence.

Use of a Lexer

Your DJ parser needs input from a DJ lexer. You have two options: use the DJ lexer you implemented in Assignment II, or use the DJ lexer provided at:
http://www.cse.usf.edu/~ligatti/compilers/21/as3/lex.yy.c

It's highly recommended that you build a DJ parser on top of your own lexer for three reasons: (1) You will have accomplished an impressive feat if at the end of the semester you have implemented dj2dism entirely by yourself. (2) Even if you scored highly on Assignment II, you may discover and fix bugs in your lexer by extending it. (3) Using a lexer that you understand and control may make it easier for you to debug your parser.

Compilation of the Parser

If you are using your own lexer, first use flex to generate a file *lex.yy.c* from your *dj.l*.

```
> flex dj.l
```

Once you have a *lex.yy.c* lexer for DJ (either by running the previous command or by downloading the provided *lex.yy.c* file), run the following commands to create and compile your DJ parser as a program called dj-parse.

```
> bison -v dj.y
> gcc dj.tab.c -o dj-parse
```

Example Executions

Many test programs appear in the DJ directory for Assignment I. However, we will test your parser on code that has not been distributed to the class.

When given one of the syntactically valid DJ programs, your parser should print nothing.

```
> ./dj-parse good1.dj
> ./dj-parse bad2.dj
>
```

To observe this "quiet" behavior while using your own lexer from Assignment I, you may need to set a flag (e.g. DEBUG_LEX) to 0 in your *dj.l* file.

When given a syntactically invalid DJ program (e.g., *bad1.dj*, *bad6.dj*, *bad13.dj*, or *bad33.dj*), your parser should print at least one accurate error message before exiting.

```
> ./dj-parse bad6.dj
Syntax error on line 4 at token ;
(This version of the compiler exits after finding the first syntax
error.)
>
```

Submission Notes

- Type the following pledge as an initial comment in your *dj.y* file: "I pledge my Honor that I have not cheated, and will not cheat, on this assignment." Type your name after the pledge. Not including this pledge will lower your grade 50%.
- Upload and submit your *dj.y* file in Canvas.
- You may submit your assignment in Canvas as many times as you like; we will grade your latest submission.
- For full credit, compilation on the cselx## machines should create no warnings or error messages.
- For this assignment, you do *not* have to add any comments to your program besides the honor pledge. Ideally, your grammar and precedence directives will make sense without further explanation. Formatting requirements are otherwise the same as on previous assignments (e.g., avoid tabs and overly long lines).