Hands-On with JavaCC

- JavaCC is Java's Compiler Compiler — it creates scanners (a.k.a. lexical analyzers, lexers) and parsers written in Java
- Since it creates both the lexer and the parser, JavaCC is thus equivalent to a combination of the traditional *lex/flex* and *yacc/bison* tools
- Free download from *https://javacc.dev.java.net*
- Pure Java, with thin (*very* thin) OS-specific scripts for convenient execution

JavaCC Pragmatics

- Unlike *yacc/bison,* JavaCC support is limited to LL(*k*) grammars, with *k* = *1* by default
- *Input:* micro- and macrosyntax specified in one *.jj* file
  - As with *lex/flex/yacc/bison*, JavaCC's *.jj* file includes the grammars as well as embedded code that defines what happens as productions are recognized
- *Output:* seven Java source files, comprising the lexer, parser, and supporting classes such as exceptions, I/O utilities, and constants

From Grammar to *.jj* File

Consider LL(1) version of the Calculator language in Figure 2.10 of the textbook:

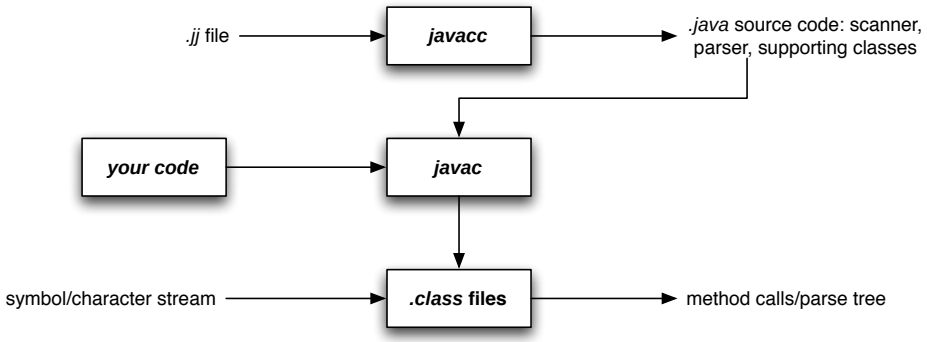
program → ***stmt_list*** $$
stmt_list → ***stmt stmt_list*** | ϵ
stmt → *id assign_op* ***expr*** | *read id* | *write* ***expr***
expr → ***term term_tail***
term_tail → ***add_op term term_tail*** | ϵ
term → ***factor factor_tail***
factor_tail → ***mult_op factor factor_tail*** | ϵ
factor → *leftParen* ***expr*** *rightParen* | *id* | *literal*
add_op → *plus* | *minus*
mult_op → *times* | *divide*

When translating the grammar into the *.jj* file:

- Define your whitespace using the SKIP token (including comments if your language has them)
- Most fixed tokens can be named directly as string literals — except when you have reserved words that "look" like identifiers
- Otherwise, define tokens in the TOKEN block, and refer to them in the CFG with < > delimiters
- For regexps, |, *, +, ?, and [ ] mean the same as in Perl; ~[*charclass*] matches whatever is *not* in *charclass*
- *.jj* CFG operators are very BNF/EBNF-like: |, *, +, and [ ] mean what you would expect

So You Spot a Production…

- Without adding further code, JavaCC's output is a parser that either: (a) ends cleanly if the input stream contains a string that belongs to the language, or (b) throws a *ParseException* or *TokenMgrError* otherwise — nice, but ultimately of limited use
- To make the parser do more:
  - Embed code in { } blocks after productions; required variables can be declared in a { } block after each non-terminal "method"
  - Use JJTree (part of JavaCC) to build an actual parse tree, or roll your own data structure, with embedded code to build it



- The sample code implements Calculator as an interpreted language, using a "virtual machine" abstraction for communication between the parser and the calculator "engine"
- For programming languages, we focus on language specification and recognition, and so won't go into JJTree — generating a parse tree and knowing what to do with it is in the realm of compiler construction

Lookahead

- By default, JavaCC expects LL(1) grammars
- When your grammar isn't LL(1), JavaCC will complain when it reaches a point where a single lookahead token isn't sufficient
- Use the LOOKAHEAD option to modify this; you can change this for the entire parser in the *.jj* options block, or you can dynamically set this at the "choice points" that require more than one lookahead token
- Check out the JavaCC Lookahead Tutorial for details

Odds & Ends

- By default, JavaCC creates a statically scoped parser; if you want to treat parsers as you would Java objects, set STATIC = false;
- Use the PARSER_BEGIN/END block to:
  - Set your parser's destination Java package
  - Define class- or instance-level variables that your embedded code needs
  - Define the external interface for the parser, if it is different from just calling the start symbol's method

LR(1) Version of Calculator

program → ***stmt_list*** $$
stmt_list → ***stmt_list stmt*** | ϵ
stmt → *id assign_op* ***expr*** | *read id* | *write* ***expr***
expr → ***term*** | ***expr add_op term***
term → ***factor*** | ***term mult_op factor***
factor → *leftParen* ***expr*** *rightParen* | *id* | *literal*
add_op → *plus* | *minus*
mult_op → *times* | *divide*

Don't bother trying to create a *.jj* definition out of this — JavaCC can tell it isn't LL(*k*), and will simply reject it