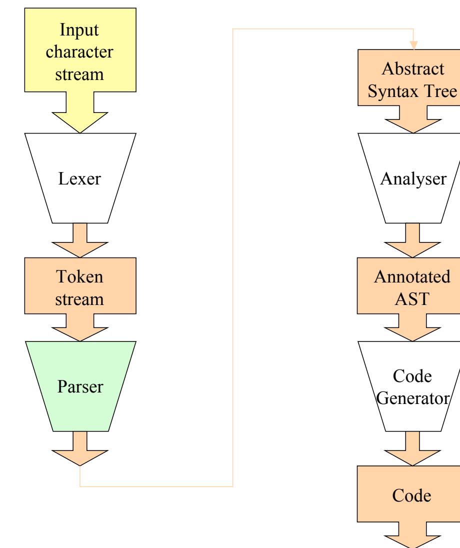
JavaCC Parser

The Compilation Task



Automated?

- The initial stages of compilation are very mechanistic
  - Lexer
    - Text to stream of tokens
      - Pattern matching
  - Parser
    - Application of grammar rules
    - Recovery from errors
  - AST Generation
    - Data structure for subsequent stages

JavaCC Parser

- JavaCC's generates a parser class
  - recursive descent parser
  - each BNF production in the .jj file is translated into a method
- Algorithm

  if the prefix of input sequence tokens match the current nonterminal definition, **then**
    remove such a prefix from the input sequence
    else throw a ParseException

An Example Grammar

```
start   ::=     expr ";"            void start() : {}
                                    {
                                      expr() ";"
                                    }

expr    ::=     (addOp)? term       void expr() : {}
                 (addOp term)*      {
                                      [addOp()] term()
                                        ( addOp() term() )*
                                    }

addOp   ::=     "+" | "-"           void addOp() : {}
                                    {
                                      "+" | "-"
                                    }
```

Implicit Token Declaration

0 or 1

Implicit Token Declaration

An Example Grammar

```
term      ::= factor                        void term() : {}
                (multop factor)*            {
                                              factor() ( multop() factor() )*
                                            }

multop    ::= "*" | "/"                     void multop() : {}
                                            {
                                              "*" | "/"
                                            }

factor    ::= primary                       void factor() : {}
                ( "**" primary)?            {
                                              primary() [ "**" primary() ]
                                            }

primary   ::= number |                      void primary() : {}
                "(" expr ")" |              {
                ident                         number()
     ( "(" rvalue ( "," rvalue )* ")" )?      | "(" expr() ")"
                                              | ident() [ "(" rvalue() ( ","
                                                rvalue())* ")" ]
                                            }
```

Java declaration section

Implicit Token Declaration

An Example Grammar

```
rvalue   ::= expr | string              void rvalue() : {}
                                        {
                                          expr() | string()
                                        }

number   ::= <INTEGER_LITERAL> |        void number() : {}
             <REAL_LITERAL>    |        {
             <HEX_LITERAL>                <INTEGER_LITERAL>
                                          | <REAL_LITERAL>
                                          | <HEX_LITERAL>
                                        }

ident    ::= <IDENTIFIER>               void ident() : {}
                                        {
                                          <IDENTIFIER>
                                        }

string   ::= <STRING>                   void string() : {}
                                        {
                                          <STRING>
                                        }
```


JavaCC Grammar Productions

- Previous slides completely define the grammar of our small expression language
- This grammar is LL(1)
- What do we do if the grammar is not LL(1)?

What do we do if the grammar is not LL(1)?

1. Identify the first and follow sets of each nonterminal symbol.
2. Prove that the grammar is LL(1)
   - for each pair of alternatives at a branch point, we need $first(A_1) \cap first(A_2) = \varnothing$
3. If the grammar is not LL(1), then transform it and goto (1).

Grammar is not LL(1)?

- If grammar is not LL(1) JavaCC will warn you
  - *Warning: Choice conflict ...*
    Consider using a lookahead of n for ...
- Choice conflict? Consider
  - `a ::= <ID> b | <ID> c`

JavaCC Look-ahead mechanism

- For alternation the first choice is the default
  - Therefore in `a ::= <ID> b | <ID> c`, the second choice is unreachable.
- Add a LOOKAHEAD specification to the first alternative, eg.

```
void a() : {} {
  LOOKAHEAD(2)
    <ID> b() |
    <ID> c()
  }
```


JavaCC Look-ahead mechanism

- If however b and c start out the same and are only distinguishable by how they end. No statically determined limit on the length of the lookahead will do.
- E.g.

```
b ::= (<NUMBER>)* ";"
c ::= (<NUMBER>)+ "." (<NUMBER>)+
```

JavaCC Look-ahead mechanism

- Use "syntactic lookahead".
  - The parser will look ahead to see if a particular *syntactic* pattern is matched before committing to a choice.

```
void a() : {} {
  // Take the first alternative if an <ID> followed by a
b()
  // appears next
   LOOKAHEAD(<ID>  b() )
     <ID> b() |
     <ID> c()
   }
```


JavaCC Look-ahead mechanism

- However the sequence `<ID> b()` may be parsed twice
  - for lookahead
  - and for regular parsing.
- Another way to resolve conflicts is to rewrite the grammar. The above nonterminal can be rewritten as

```
a  ::= <ID> (b | c)
```

- So do NOT use LOOKAHEAD indiscriminately.

JavaCC Semantic Lookahead

- Two types of look-ahead mechanisms
  - Syntactic
    - A particular token is looked ahead in the input stream.
  - Semantic
    - Any arbitrary Boolean expression can be specified as a lookahead parameter.

JavaCC Semantic Lookahead

- Example
  - `A :: aBc and B :: b(c)?`
    - Valid strings: “abc” and “abcc”

```
void B() : {}
{
  "b"
  [ LOOKAHEAD(getToken(1).kind == C && getToken(2).kind !=
  C )
    <C:"c">
  ]
}
```

JavaCC Lookahead Summary

- Exploration of tokens further ahead in the input stream.
- Backtracking is unacceptable due to performance hit.
- By default JavaCC has 1 token look-ahead. Can specify any number for look-ahead.
  - Globally and locally

How does JavaCC differ from standard LL(1) parsing?

- JavaCC is more flexible
- It lets you use multiple token lookahead
  - syntactic lookahead
  - semantic lookahead.
- Without lookahead
  - only subtly different from LL(1) parsing

Finally

- Since LL(1) ⊂ LALR(1) ⊂ LR(1), wouldn't a tool based on LALR or LR parsing be better?
  - True, but a hard problem
- JavaCC is based on LL(1) parsing, but it allows you to use grammars that are not LL(1).
- JavaCC can handle any grammar that is not left-recursive.
- If necessary use the ambiguous set of grammar rules, and use other mechanisms to resolve the ambiguity.

Building and Executing

```
C:\Rob\jj>javacc Expression.jj
Java Compiler Compiler Version 3.0 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file Expression.jj . . .
Parser generated successfully.

C:\Rob\jj>javac *.java

C:\Rob\jj>java Expression
Reading from standard input...
-2+p("fred",3/4);
Thank you.
```

Questions

- Write a grammar for the Lexer portion of JavaCC
- What happens is the input language is incorrect?