Compilers

ANTLR

ANTLR

- An ANTLR input file (.g) has entries for
  - Headers
  - Options
  - Class Definitions
  - Rules
- These can be repeated for multiple
  - Lexers, parsers, and tree walkers

Input File Schema

```
header { ... }// global
options { ... } // global
{... } // class preamble
class <name> extends <builtin name>;
options { ... } // class specific
tokens { ... }
<list of rules>
... more class definitions ...
```

Simple Example

```
class ExprParser extends Parser;
options {defaultErrorHandler=false;}
expr : mexpr ((PLUS|MINUS) mexpr)* ;
mexpr: atom (TIMES atom)* ;
atom : INT | LPAREN expr RPAREN ;
```

```
class ExprLexer extends Lexer;
options { k=2; }
LPAREN: '(' ;
RPAREN: ')' ;
PLUS  : '+' ;
MINUS : '-' ;
TIMES : '*' ;
INT   : ('0'..'9')+ ;
WS    : ( ' ' | '\r' '\n' | '\n' | '\t' )
          {$setType(Token.SKIP);}
      ;
```

Simple Example

```
class ExprParser extends Parser;
options {defaultErrorHandler=false;}
expr : mexpr ((PLUS|MINUS) mexpr)* ;
mexpr: atom (TIMES atom)* ;
atom : INT | LPAREN expr RPAREN ;
```

```
class ExprLexer extends Lexer;
options { k=2; }
LPAREN: '(' ;
RPAREN: ')' ;
PLUS  : '+' ;
MINUS : '-' ;
TIMES : '*' ;
INT   : ('0'..'9')+ ;
WS    : ( ' ' | '\r' '\n' | '\n' | '\t' )
        {$setType(Token.SKIP);}
      ;
```

Generated Parser

- ANTLR generates
  - A predictive recursive descent parser
  - Grammar must have all the LL(1) properties
    - e.g., no left recursion, no common prefixes, no ambiguity in parse table
    - Can relax some of this through use of special directives
- Structure of parser matches grammar closely

```
expr : mexpr ((PLUS|MINUS) mexpr)* ;
```

```
public class ExprParser extends antlr.LLkParser ... {
  ...
  public final void expr() {
    mexpr();
    _loop2074:
    do {
      if ((LA(1)==PLUS||LA(1)==MINUS)) {
        switch ( LA(1)) {
          case PLUS: match(PLUS); break;
          case MINUS: match(MINUS); break;
        }
        mexpr();
      } else {
        break _loop2074;
      }
     } while (true);
  }
  ...
}
```

Headers

- Global header is inserted at the top of all generated files
  - Useful for front-end wide imports
- Class preamble is inserted before class declaration in generated file
  - Useful for phase-specific imports

Sample Headers : input

```
header {
        // global comment
}


{
        // lexer preamble comment
}
class HeaderLexer extends Lexer;
...


{
        // parser preamble comment
}
class HeaderParser extends Parser;
...
```

Sample Headers : output

```
// $ANTLR : "header.g" -> "HeaderLexer.java"$
        // global comment
import ...
        // lexer class preamble
public class HeaderLexer extends antlr.CharScanner  ... { ...  }
```

```
// $ANTLR : "header.g" -> "HeaderParser.java"$
        // global comment
import ...
        // parser class preamble
public class HeaderParser extends antlr.LLkParser ... { ... }
```

```
// $ANTLR : "header.g" -> "HeaderLexerTokenTypes.java"$
        // global comment
public interface HeaderLexerTokenTypes { ... }
```

Options

Some common options

- k – tokens of lookahead
- package – package to put generated files in
- defaultErrorHandler – toggle default off or on
- filter – pass through input not matched by lexer

Lots of others for

- debugging, verbose output, class name control, ...

Rules

General form of a rule is:

```
rulename [args] returns [retval]
    options { local rule options }
    { optional initialization code }
    :   alternative_1
    |   alternative_2
    ...
    |   alternative_n
    ;
```

Rules : Generated Code

Conceptually a rule generates a method

```
<retval> rule(<args>) {
    <initialization code>
    ... matcher for alternatives ...
}
```

Alternatives

A sequence of

- Token and rule name expressions
  - Concatenate (' '), '|', '*', '+', '?', '..', '.', '~'
- Semantic actions
  - Fragments of code contained in '{' '}'
  - Executed in order in the parse
  - Can be nested within rule structure

```
( {do this every time}:
  {action for first alternative} alt1 |
  {action for second alternative} alt2
)*
```

Production element labels

Actions refer to matched elements by name

```
assign :    v:ID “=” expr ;
            { System.out.println(“assign to ”+v.getText()); }
```

Lots of examples of this in the expression evaluation examples in the course CVS repository.

Syntactic Predicates

- One can control lookahead globally by setting option k
- It can also be convenient to perform customized rule disambiguation using

```
( lookahead production) => production
```

- when the `lookahead production` matches then continue the parse with `production`
  - `lookahead production` cannot have actions

Semantic Predicates

- Sometimes we can't decide how to continue the parse until we see the input
- We use a special boolean valued action

  { boolean predicate code }?

  that is evaluated at run-time at it's position in the parse
- Validating predicates can appear anywhere except the beginning of a rule
  - They signal a SemanticException

Disambiguating Predicates

- Appear at the beginning of a production

```
stat :
        { isTypeName(LT(1)) }? ID ID “;”
      |  ID “=” expr “;”
      ;
```

- Need to use LT here since we haven't matched the token and cannot access it by name

SJC Grammar

- I'll walk you through some excerpts of the SJC lexer and parser specs
- Note that the .g file will be modified substantially when we consider the semantic actions

```
header {
package sjc.parser;
...
}

/**
* StaticJava parser.
* This class is automatically generated by ANTLR.
*
* @author <a href="mailto:robby@cis.ksu.edu">Robby</a>
*/
class SJParser extends Parser;

options { k = 2; }

compilationUnit
        :       classDefinition
                EOF
        ;
```

```
classDefinition
        :       "public" "class" IDENT LCURLY
                mainMethodDeclaration
                ( ("static" type IDENT SEMI) =>
                        fieldDeclaration
                |       methodDeclaration )*
                RCURLY
        ;

...

methodDeclaration
        :       "static" returnType IDENT
                LPAREN ( params )? RPAREN
                LCURLY methodBody RCURLY
        ;
```

```
params
        :       param ( COMMA param )*
        ;

param
        :       type IDENT
        ;

methodBody
{
        boolean hasSeenStatement = false;
}
        :       (
                        (type IDENT SEMI) =>
                        { !hasSeenStatement }?
                        localDeclaration
                |       statement
                        { hasSeenStatement = true;}
                )*
        ;
```

```
ifStatement
        :       "if" LPAREN exp RPAREN
                LCURLY ( statement )* RCURLY
                ( "else" LCURLY ( statement )*
                RCURLY )?
        ;


relationalExp
        :       additiveExp ( ( LT | GT | LE | GE ) additiveExp )*
        ;


primaryExp
        :       n:NUM_INT
                { new BigInteger(n.getText()).bitLength() < 32 }?
        |       ...
        ;
```