Compilers

ANTLR

ANTLR

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

Input File Schema

```
grammar <name>;
options { ... }
@header {... } // global
@lexer:header {... } // lexer-specific
@members {... }
<list of rules>
```

We’ll consider “combined” grammars here

Simple Example

```
grammar Example2;


options { backtrack=true; }
@header { package antlr.example2; }
@lexer::header { package antlr.example2; }
```

```
start    : e EOF ;
e        : t '+' e | t ;
t        : INT | INT '*' t | '(' e ')' ;


INT      : ( '0' | ('1'..'9') ('0'..'9')* ) ;
WS       : (' '|'\r'|'\t'|'\u000C'|'\n') { $channel=HIDDEN; } ;
```

Simple Example

```
grammar Example2;

options { backtrack=true; }
@header { package antlr.example2; }
@lexer::header { package antlr.example2; }


start   : e EOF ;
e       : t '+' e | t ;
t       : INT | INT '*' t | '(' e ')' ;


INT     : ( '0' | ('1'..'9') ('0'..'9')* ) ;
WS      : (' '|'\r'|'\t'|'\u000C'|'\n') { $channel=HIDDEN; } ;
```

Simple Example

```
grammar Example2;

options { backtrack=true; }
@header { package antlr.example2; }
@lexer::header { package antlr.example2; }


start   : e EOF ;
e       : t '+' e | t ;
t       : INT | INT '*' t | '(' e ')' ;


INT     : ( '0' | ('1'..'9') ('0'..'9')* ) ;
WS      : (' '|'\r'|'\t'|'\u000C'|'\n') { $channel=HIDDEN; } ;
```

Generated Parser

- ANTLR generates
  - A predictive recursive descent parser
- In v2 grammar required LL(1) properties
  - e.g., no left recursion, no common prefixes, no ambiguity in parse table
  - Special directives allowed this to be relaxed
- v3 supports the more powerful LL(*)
  - Better lookahead and backtracking lead to very straightforward grammar specs

e : t '+' e | t ;

```
public final void e() throws RecognitionException {
    try {
        // Predict rule alternative to use

        // Match rule and execute actions

    } catch (RecognitionException re) {
        reportError(re);
        recover(input,re);
    } finally {
        if ( backtracking>0 )
            memoize(input, 2, e_StartIndex);
    }
}
```

e : t '+' e | t ;

```
// Predict rule alternative to use
int LA1_0 = input.LA(1);
if ( (LA1_0==INT) ) { // LA(1) == 'INT'
    switch ( input.LA(2) ) {
    case 7: // LA(1..2) == 'INT *'
        int LA1_3 = input.LA(3);
        if ( (LA1_3==INT) ) { // LA(1..3) == 'INT * INT'
            switch ( input.LA(4) ) {
            case 9: // LA(1..4) == "INT * INT )"
                alt1=2; ...
            case 6: // LA(1..4) == "INT * INT +"
                alt1=1; ...
            }
        } else if ( (LA1_3==8) ) { // LA(1..3) == "INT * ("
            alt1=2;
        }
     case 9: // LA(1..2) == 'INT )'
         alt1=2; ...
     case 6: // LA(1..2) == 'INT +'
         alt1=1; ...
     }
}
```

e : t '+' e | t ;

```
// Match rule and execute actions
switch (alt1) {
    case 1 : // e : t '+' e
        pushFollow(FOLLOW_t_in_e53);
        t();
        popFollow(); if (failed) return ;

        match(input,6,FOLLOW_6_in_e55); if (failed) return ;

        pushFollow(FOLLOW_e_in_e57);
        e();
        popFollow(); if (failed) return ;

    case 2 : // e : t
        pushFollow(FOLLOW_t_in_e62);
        t();
        popFollow(); if (failed) return ;
}
```

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

Options

Some common options

- language - controls target language, default is Java
- backtrack - generates backtracking parser
- memoize - crucial performance option when backtracking
- k - if you don’t want LL(*) set k to lookahead

Lots of other options … poke around

Rules

General form of a rule is:

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

Implicit rule organization

- Rule name starts with capital letter
  - Defines a lexical rule - a token class
- Rule name starts with lowercase letter
  - Defines a grammar production
- ANTLR will separate implementations into
  - <name>Lexer.java
  - <name>Parser.java

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 SVN repository.

Syntactic Predicates

- One can default lookahead prediction
- 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

```
grammar StaticJava;

options { backtrack=true; memoize=true; }

@header {
package sjc.parser;

import java.math.BigInteger;

/**
 * StaticJava parser.
 *
 * @author robby
 */
}

compilationUnit
        :       classDefinition
                EOF
        ;
```

```
classDefinition
        :       'public' 'class' ID '{'
                mainMethodDeclaration
                ( fieldDeclaration
                |       methodDeclaration )*
                '}'
        ;

…

methodDeclaration
        :       'static' returnType ID
                '(' ( params )? ')'
                '{' methodBody '}'
```

```
params
        :       param ( ',' param )*
        ;

param
        :       type ID
        ;

methodBody
        :       localDeclaration*
                statement*
        ;
```

```
methodBody
        :       localDeclaration*
                statement*
        ;
```

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

V2

```
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 }?
        |       ...
        ;
```