ANTLR Grammar for Esterel

ANTLR

Esterel.g

public class EsterelParser extends Parser;

file : expr EOF!;

class EsterelLexer extends Lexer;

ID : LETTER (LETTER| DIGIT)* ;

ANTLR Lexer Specifications

Look like

class MyLexer extends Lexer;
'options {
    option = value
}

Token1 : 'char' 'char';
Token2 : 'char' 'char';
Token3 : 'char' ('char')?;

Tries to match all non-protected tokens at once.

ANTLR Parser Specifications

Look like

class MyParser extends Parser;
'options {
    option = value
}

rule1 : Token1 Token2
    | Token3 rule2 ;
rule2 : (Token1 Token2)* ;
rule3 : rule1

Looks at the next $k$ tokens when deciding which option to consider next.

ANTLR

EsterelParser.java

public class EsterelParser extends Parser;

file : expr EOF!;

class EsterelLexer extends Lexer;

ID : LETTER (LETTER| DIGIT)* ;

ANTLR Lexer Specifications

Look like

class MyLexer extends Lexer;
'options {
    option = value
}

Token1 : 'char' 'char';
Token2 : 'char' 'char';
Token3 : 'char' ('char')?;

Tries to match all non-protected tokens at once.

ANTLR Parser Specifications

Look like

class MyParser extends Parser;
'options {
    option = value
}

rule1 : Token1 Token2
    | Token3 rule2 ;
rule2 : (Token1 Token2)* ;
rule3 : rule1

Looks at the next $k$ tokens when deciding which option to consider next.

ANTLR

EsterelLexer.java

public class EsterelLexer extends Lexer;

ID : LETTER (LETTER| DIGIT)* ;

ANTLR Lexer Specifications

Look like

class MyLexer extends Lexer;
'options {
    option = value
}

Token1 : 'char' 'char';
Token2 : 'char' 'char';
Token3 : 'char' ('char')?;

Tries to match all non-protected tokens at once.

ANTLR Parser Specifications

Look like

class MyParser extends Parser;
'options {
    option = value
}

rule1 : Token1 Token2
    | Token3 rule2 ;
rule2 : (Token1 Token2)* ;
rule3 : rule1

Looks at the next $k$ tokens when deciding which option to consider next.

ANTLR

EsterelParser.java

public class EsterelParser extends Parser;

file : expr EOF!;

class EsterelLexer extends Lexer;

ID : LETTER (LETTER| DIGIT)* ;

ANTLR Lexer Specifications

Look like

class MyLexer extends Lexer;
'options {
    option = value
}

Token1 : 'char' 'char';
Token2 : 'char' 'char';
Token3 : 'char' ('char')?;

Tries to match all non-protected tokens at once.

ANTLR Parser Specifications

Look like

class MyParser extends Parser;
'options {
    option = value
}

rule1 : Token1 Token2
    | Token3 rule2 ;
rule2 : (Token1 Token2)* ;
rule3 : rule1

Looks at the next $k$ tokens when deciding which option to consider next.

ANTLR

EsterelLexer.java

public class EsterelLexer extends Lexer;

ID : LETTER (LETTER| DIGIT)* ;

ANTLR Lexer Specifications

Look like

class MyLexer extends Lexer;
'options {
    option = value
}

Token1 : 'char' 'char';
Token2 : 'char' 'char';
Token3 : 'char' ('char')?;

Tries to match all non-protected tokens at once.

ANTLR Parser Specifications

Look like

class MyParser extends Parser;
'options {
    option = value
}

rule1 : Token1 Token2
    | Token3 rule2 ;
rule2 : (Token1 Token2)* ;
rule3 : rule1

Looks at the next $k$ tokens when deciding which option to consider next.

ANTLR

EsterelParser.java

public class EsterelParser extends Parser;

file : expr EOF!;

class EsterelLexer extends Lexer;

ID : LETTER (LETTER| DIGIT)* ;

ANTLR Lexer Specifications

Look like

class MyLexer extends Lexer;
'options {
    option = value
}

Token1 : 'char' 'char';
Token2 : 'char' 'char';
Token3 : 'char' ('char')?;

Tries to match all non-protected tokens at once.
A Lexer for Esterel

Identifiers are standard:

\[ \text{ID} \colon ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'| '_' | '0'..'9')^* ; \]

String constants must be contained on a single line and may contain double quotes, e.g.,

"This is a constant with \"double quotes\"

ANTLR makes this easy: annotating characters with \! discards them from the token text:

\[ \text{StringConstant} \colon '"'! ( ˜('"' | '\n')| ('"'! '"'))* '"'!; \]

I got in trouble with the ˜ operator, which inverts a character class. Invert with respect to what?

Needed to change options:

\[
\text{options} \{
\begin{array}{l}
\text{charVocabulary} = '\\3'..'\377';
\text{exportVocab} = \text{Esterel};
\end{array}
\}
\]

I split numbers into Number and FractionalNumber to avoid this problem: If the two rules were combined, the lookahead set for Number would include a period (e.g., from "1") followed by end-of-token (e.g., from "1 by itself").

Example numbers:

Number Rules Continued

\[ \text{Number} \colon ('0'..'9')^+ (.' ('0'..'9')^* (Exponent)? ( ('f'|'F') \{ $\text{setType} = \text{FloatConst}; \} | /* empty */ \{ $\text{setType} = \text{DoubleConst}; \} | /* empty */ \} | / * empty */ \{ $\text{setType} = \text{Integer}; \} | / * empty */ \}
\]

\[ \text{FractionalNumber} \colon . ('0'..'9')^+ (Exponent)? ( ('f'|'F') \{ $\text{setType} = \text{FloatConst}; \} | /* empty */ \{ $\text{setType} = \text{DoubleConst}; \} ) \]

\[ \text{protected Exponent} \colon (e'|E') (\+'-')? ('0'..'9')^+; \]

```
\begin{array}{|c|c|}
\hline
\text{Numbers} & \text{Comments} \\
\hline
\text{12.3} & \text{Simple comments start with % and end at end-of-line.} \\
\text{.123} & \text{Multiple-line comments start with % and end with } % \\
\text{1.23E1} & \text{Multiple-line comments start with % and end with } % \\
\text{1.23E1f} & \text{} \\
\end{array}
```

```
\begin{array}{|c|c|}
\hline
\text{.1$.21$} & \text{} \\
\text{First} & \text{Second} \\
\hline
\text{1} & \text{.2} \\
\text{.1} & \text{1} \\
\end{array}
```

```
\begin{array}{|c|c|}
\hline
\text{Number} & \text{Number} \\
\hline
\text{Opt} & \text{Opt} \\
\text{0} & \text{0} \\
\text{9} & \text{9} \\
\end{array}
```

```
\begin{array}{|c|c|}
\hline
\text{FractionalNumber} & \text{FractionalNumber} \\
\hline
\text{Opt} & \text{Opt} \\
\text{0} & \text{0} \\
\text{9} & \text{9} \\
\end{array}
```

```
\begin{array}{|c|c|}
\hline
\text{Exponent} & \text{Exponent} \\
\hline
\text{Opt} & \text{Opt} \\
\text{0} & \text{0} \\
\text{9} & \text{9} \\
\end{array}
```

```
\begin{array}{|c|c|}
\hline
\text{FractioNumber} & \text{FractioNumber} \\
\hline
\text{Opt} & \text{Opt} \\
\text{0} & \text{0} \\
\text{9} & \text{9} \\
\end{array}
```

```
\begin{array}{|c|c|}
\hline
\text{Exponent} & \text{Exponent} \\
\hline
\text{Opt} & \text{Opt} \\
\text{0} & \text{0} \\
\text{9} & \text{9} \\
\end{array}
```

```
\begin{array}{|c|c|}
\hline
\text{Exponent} & \text{Exponent} \\
\hline
\text{Opt} & \text{Opt} \\
\text{0} & \text{0} \\
\text{9} & \text{9} \\
\end{array}
```

```
\begin{array}{|c|c|}
\hline
\text{FractioNumber} & \text{FractioNumber} \\
\hline
\text{Opt} & \text{Opt} \\
\text{0} & \text{0} \\
\text{9} & \text{9} \\
\end{array}
```

```
\begin{array}{|c|c|}
\hline
\text{Exponent} & \text{Exponent} \\
\hline
\text{Opt} & \text{Opt} \\
\text{0} & \text{0} \\
\text{9} & \text{9} \\
\end{array}
```
A Parser for Esterel

Esterel’s syntax started out using ; as a separator and later allowed it to be a terminator. The language reference manual doesn’t agree with what the compiler accepts.

Grammar from the LRM

But in fact, the compiler accepts

module TestSemicolon1:
  nothing;
end module
module TestSemicolon2:
  nothing; nothing;
end module
module TestSemicolon3:
  nothing; nothing
end module

Rule seems to be “one or more statements separated by semicolons except for the last, which is optional.”

Grammar for Statement Sequences

Obvious solution:

sequence : atomicStatement
  (SEMICOLON atomicStatement) *
  (SEMICOLON)? ;

warning: nondeterminism upon
k==1:SEMICOLON
between alt 1 and exit branch of block
Which option do you take when there’s a semicolon?

Solution: tell ANTLR to be greedy and prefer the iteration solution.

sequence : atomicStatement
  (options {greedy=true;})
  : SEMICOLON! atomicStatement }
  (SEMICOLON)!?

Nondeterminism

Delays can be “A“ “X A” “immediate A” or “[A and B].”

delay : expr bSigExpr
  | bSigExpr
  | "immediate" bSigExpr ;
  bSigExpr : ID
  | "[" signalExpression "]" ;
  expr : ID | /* ... */ ;
Which choice when next token is an ID?
Nondeterminism

delay : expr bSigExpr |
| "immediate" bSigExpr;

What do we really want here?
If the delay is of the form "expr bSigExpr," parse it that way.
Otherwise try the others.

delay : ( (expr bSigExpr) => delayPair |
| bSigExpr |
| "immediate" bSigExpr ) ;
delayPair : expr bSigExpr ;
The => operator means "try to parse this first. If it works,
choose this alternative."

Greedy Rules

The author of ANTLR writes
I have yet to see a case when building a parser
grammar where I did not want a subrule to match
as much input as possible.

However, it is particularly useful in scanners:

COMMENT : "/ * " (.) * " * /" ;
This doesn't work like you'd expect...

Turning Off Greedy Rules

The right way is to disable greedy:

COMMENT : "/ * " 
(options {greedy=false;}) : 
" */ " ;
This only works if you have two characters of lookahead:
class L extends Lexer;
options { k=2; }

CMT : "/ * " (options {greedy=false;}) ; 
" */ " ;

Removing the Warning

class MyGram extends Parser;

stmt : "if" expr "then" stmt ("else" stmt)? ;

A Simpler Language

class MyGram extends Parser;

match(LITERAL_if);
expr();
match(LITERAL_then);
stmt();
switch (LA(1)) {
case LITERAL_else:
stmt();
break;
case LITERAL_fi:
break;
default:
throw new SyntaxError(LT(1));
}
match(LITERAL_fi);