ANTLR and Tiger
COMS W4115
Prof. Stephen A. Edwards
Spring 2002
Columbia University
Department of Computer Science

ANTLR

Tiger.g
class TigerParser extends Parser;
file : expr EOF!;

TigerLexer.java
public class TigerLexer 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 Parser Specifications

An ANTLR grammar for Esterel
Esterel: Language out of France. Programs look like
module ABRO:
  input A, B, R;
  output O;
loop
  [ await A || await B ];
  emit O
each R
end module

ANTLR Lexer Specifications

A Lexer for Esterel
Operators from the language reference manual:
. # + - / || < > , = ; := ( )
[ ] ?? <= >= <> =>
Main observation: none longer than two characters. Need k = 2 to disambiguate, e.g., ? and ??.
class EsterelLexer extends Lexer;
options {
    k = 2;
}

A Lexer for Esterel
Identifiers are standard:
ID :
  ('a'..'z' | 'A'..'Z')
  ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*

A Lexer for Esterel
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:
StringConstant :
  '"'!
  \"\n" ! \"\n" ! \n" ! \"" ! \n" ! \"" ! \n"

A Lexer for Esterel
Next, I wrote a rule for each punctuation character:
PERIOD :
  ";
POUND :
  ";
PLUS :
  ";
DASH :
  ";
SLASH :
  ";
STAR :
  ";
PARALLEL :
  ";

A Lexer for Esterel

A Lexer for Esterel
A Lexer for Esterel

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

Needed to change options:

```java
options {
    k = 2;
    charVocabulary = '\3'..'\377';
    exportVocab = Esterel;
}
```

A Lexer for Esterel

Another problem: ANTLR scanners check each recognized token's text against keywords by default. A string such as "abort" would scan as a keyword!

```java
options {
    k = 2;
    charVocabulary = '\3'..'\377';
    exportVocab = Esterel;
    testLiterals = false;
}
```

IDoptions { testLiterals = true; }

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

NonParallel:

AtomicStatement

Sequence

Sequence:

  SequenceWithoutTerminator : opt

SequenceWithoutTerminator:

  AtomicStatement ; AtomicStatement

AtomicStatement:

  nothing pause ...

Grammar from the LRM

But in fact, the compiler accepts

```java
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."

Nondeterminism

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

Is equivalent to

```java
sequence : atomicStatement seq1 seq2 ;
seq1 : SEMICOLON atomicStatement seq1
  | /* nothing */ ;
seq2 : SEMICOLON
  | /* nothing */ ;
```

Nondeterminism

```java
sequence : atomicStatement seq1 seq2 ;
seq1 : SEMICOLON atomicStatement seq1
  | /* nothing */ ;
seq2 : SEMICOLON
  | /* nothing */ ;
```

How does it choose an alternative in seq1?

First choice: next token is a semicolon.
Second choice: next token is one that may follow seq1.
But this may also be a semicolon!

Nondeterminism

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

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

Grammar for Statement Sequences

Obvious solution:

```java
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?
Nondeterminism

Delays can be “A” “X” “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) => delayPair
  | bSigExpr
  | "immediate" bSigExpr ) ;

delayPair : expr bSigExpr ;
```

The => operator means “try to parse this first. If it works, choose this alternative.”

Tiger Expressions

```
"hello"
1024
nil
foo
-(1+2)
1 * 3
foo := 10
bar(10,20)
(a := 5; b := 3; c := 2)
if a then 20 else 30
while a < 5 do a := a + 1
for i := 1 to 5 do i + 2
break
let ... in ... end
```

Ivalues

```
“Something that may appear on the left side of an assignment”

lvalue:
  id
  lvalue : id
  lvalue [ expr ]

foo := 3
bar.baz := 5
biff[10] := 20
derf[5].x := 3
```

Object Constructors

Tiger has array and record types.

```
let
  type ia = array of integer
  type point = { x : integer, y : integer }
  var a := ia [5] of 0
  var p := point { x = 1, y = 2 }
in
  0
end
```

Tiger AST

```
lvalue
  : ID
  | #( FIELD lvalue ID ) // lvalue.field
  | #( SUBSCRIPT lvalue expr) // lvalue[expr]
 ;

expr
  : "nil"
  | lvalue
  | STRING
  | NUMBER
  | #( NEG expr ) // - e
  | #( BINOP expr expr ) // e+e, e*e
```

Tiger AST

```
#( ASSIGN lvalue expr ) // l := e
#( CALL ID (expr)* ) // f(e, e)
#( SEQ (expr)* ) // (e ; e)
#( RECORD ID // t { a=b, c=d }
    #(FIELD ID expr))*
#( NEWARRAY ID expr expr ) // t [e] of e
#( "if" expr expr (expr)? )
#( "while" expr expr )
#( "for" ID expr expr expr )
"break"
#( "let" #(DECLS (#(DECLS (decl) + )))* ) expr )
```
Tiger AST

decl
  : #( "type" ID type )
    | #( "var" ID (ID | "nil") expr )
    | #( "function" ID fields (ID | "nil") expr )

;

type
  : ID
    | fields   // a:b, c:d
    | #( "array" ID )   // array of type

;

fields : #( FIELDS ( #(FIELD ID) )* ) ;