Language Engineering Coursework & ANTLR Tutorial

Rob Frampton

October 23, 2014

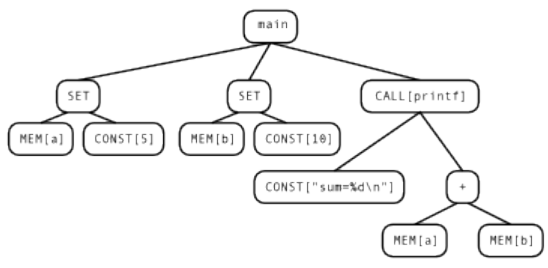
Parsing

- "A parser is a software component that takes input data (frequently text) and builds a data structure [...] giving a structural representation of the input, checking for correct syntax in the process. The parsing may be preceded or followed by other steps."

Compilers

A compiler translates a source language (e.g. **high-level programming language**) into a target language (e.g. **assembly language**).

```
int main() {
  int a = 5;
  int b = 10;
  printf("sum=%d\n", a + b);
}
```




```
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    $5, -8(%rbp)
    movl    $10, -4(%rbp)
    movl    -4(%rbp), %eax
    movl    -8(%rbp), %edx
    addl    %eax, %edx
    movl    $.LC0, %eax
    movl    %edx, %esi
    movq    %rax, %rdi
    movl    $0, %eax
    call    printf
```

Parser generators

Given the source language grammar, how do we write a parser for our compiler? We could:

- Write a program specifically for the grammar
  - Fast and flexible, but time-consuming
- Use existing software which can interpret a grammar description
  - Easy, but inflexible and slower
- Use existing software which generates a specific program given a grammar description
  - i.e. a *parser generator* or *compiler-compiler*
  - Easier, fast and flexible

ANTLR: Overview

In this coursework we suggest a parser generator written in and targeting Java, called ANTLR.

Development:

$$\text{MyParser.g} \xrightarrow{antlr3} \text{MyParser.java} \xrightarrow{javac} \text{MyParser.class}$$

Runtime:

$$\text{myprogram.w} \xrightarrow{java\ MyParser} \text{myprogram.ass} \xrightarrow{assmule} \text{emulation}$$

ANTLR

ANTLR

ANTLR generate three types of parser:

- **Lexer**: character stream to token stream
- **Parser**: token stream to syntax tree
- **Tree Parser**: reads a syntax tree

In this coursework we are only interested in the first two.

ANTLR Lexer

Grammar file begins with:

```
lexer grammar Lex;
```

Followed by a set of parse rules, e.g.:

```
SEMICOLON  : ';' ;

WRITELN    : 'writeln' ;

INTNUM     : ('0'..'9')+ ;

STRING     : '\'' ('\'' '\'' | ~'\'')* '\'';
```

Lexer rule names should begin with an upper case character.

ANTLR Lexer

Rules may include:

- Character constants
- Parentheses
- Another rule name
- Alternatives (|), range (..), not (~)
- Optional (?), zero-or-more (*), one-or-more (+)

```
WRITELN     : 'writeln' ;

INTNUM      : ('0'..'9')+ ;

STRING      : '\'' ('\'' '\'' | ~'\'')* '\'';
```

ANTLR Lexer

Watch out for ambiguity, rule order matters:

```
IDENTIFIER    : ('0'..'9' | 'a'..'z' | 'A'..'Z' | '_')+ ;

INTNUM        : ('0'..'9')+ ;
```

ANTLR Lexer

If we want extra functionality, we can enter Java code into our rules.

```
@members {
  int lineCount = 0;
}

...

NEWLINE  : ('\r' | '\n') { lineCount++; } ;

WS       : (' ' | '\t')+ {skip();} ;
```

ANTLR Parser

Grammar file begins with:

```
parser grammar Syn;
```

Followed by a set of parse rules, e.g.:

```
statements :
    statement ( SEMICOLON^ statement )*
  ;

statement :
    WRITE^ OPENPAREN! ( INTNUM | string ) CLOSEPAREN!
  | WRITELN
  ;
```

Parser rule names should begin with an lower case character.

ANTLR Parser

For parser rules we have some extra syntax for tree construction:

- ^ on a token makes it the root
- ! on a token ignores it

```
statements :
    statement ( SEMICOLON^ statement )*
  ;

statement :
    WRITE^ OPENPAREN! ( INTNUM | string ) CLOSEPAREN!
  | WRITELN
  ;
```

ANTLR Parser

Again, we can insert Java code into our rules:

```
expression:
    ( m=MINUS^ )? term
    { if ($m != null) $m.setType(UMINUS); }
```

We can capture tokens into variables which are accessed in the Java code using the $ syntax. Tokens are of type `org.antlr.runtime.Token`.

See: http://www.antlr3.org/api/Java/org/antlr/runtime/Token.html

Syntactic predicates

Sometimes we encounter grammars which cannot be expressed using an LL(*) parser. We can force ANTLR to 'look ahead' and then 'backtrack' in order to force it to choose an alternative.

```
statement:
    ^( IF condition compound )
  | ^( IF condition compound ELSE compound )
  ;

statement:
    ( IF condition compound ) =>
      ^( IF condition compound)
  | ( IF condition compound ELSE compound ) =>
      ^( IF condition compound ELSE compound )
  ;
```

Common Tree

- This covers the lexical analysis (lexer) and syntactic analysis (parser) stages of your compiler.
- The next stages of your compiler are written in Java manually.
- You can walk the syntax tree using the ANTLR CommonTree object.
  See: http://www.antlr3.org/api/Java/org/antlr/runtime/tree/CommonTree.html

```
public static void program(CommonTree ast, IRTree irt)
{
  statements(ast, irt);
}

public static void statements(CommonTree ast, IRTree irt)
{
  Token t = ast.getToken();
  ...
}
```