Homework 2: Parser and Lexer

Remi Meier
Compiler Design – 08.10.2015
Compiler phases

Front-end
  - Javali
  - Lexical Analysis
  - Syntactic Analysis

Optimizations
  - IR
  - AST

Back-end
  - IR
  - Machine dependent

Compiler
  - Machine independent

x86 Assembly

Semantic Analysis
Homework 2

How do we...
- check if a program follows the syntax of Javali?
- extract meaning / structure?
Homework 2

Part 2b

Lexer ➔ Token Stream ➔ Parser ➔ Parse Tree ➔ Javali AST

Part 2a

Part 2b

4
Lexical Analysis

Lexer
- Read input character by character
- Recognize character groups → tokens

Token
- Sequence of characters with a collective meaning → grammar terminals
- E.g. constants, identifiers, keywords, ...
Lexical Analysis

```java
class Main {
    void main() {
        write(222);
        writeln();
    }
}
```

Token stream:

```
ID: class  ID: Main  MISC: {  ID: void  ID: main  MISC: (  MISC: )  ... 
```
Syntactic Analysis

Parser

- **Check** if token stream follows the grammar
- Group tokens hierarchically (**extract structure**) → Parse Tree / Abstract Syntax Tree
TOP-DOWN PARSER
Top-down parsers

Grammar in Extended Backus-Naur Form (EBNF):

```plaintext
statement: 
  return 
  | assign

return: 
  'return' expr ';'

assign: 
  ID '==' expr ';'

expr: ID '+' ID
```

```
return a + b ;
```

```
statement
  return
  'return' expr ';
  'a' '+' 'b'
```
Implementation

Grammar in Extended Backus-Naur Form (EBNF):

```
statement: return | assign

return: 'return' expr ';'

assign: ID '=' expr ';'

expr: ID '+' ID
```

How to deal with alternatives?
Lookahead

Grammar in Extended Backus-Naur Form (EBNF):

```
statement:  
    return  
    | assign

return:    
    'return' expr ';

assign:    
    ID '=' expr ';

eexpr:    ID '+' ID
```

```c
void statement() {
    if (next() is 'return') {
        return();
    } else if (next() is ID) {
        assign();
    }
}
```
http://www.antlr4.org/
(or HW2 fragment)
ANTLR

Token specifications + Grammar

Top-down parser generator
• ALL(*) adaptive, arbitrary lookahead
• handles any non-left-recursive context-free grammar

MyLexer.java
MyParser.java
ANTLR – Grammar description

Start rule matching end-of-file

Lower-case initial: Parser

Literals → Tokens

Upper-case initial: Lexer

/* This is an example */
grammar Example;

/* Parser rules = Non-terminals */
program :
    statement* EOF ;

statement :
    assignment ;
    | expression ;

/* Lexer rules = Terminals */
Identifier : Letter (Letter | Digit)* ;
Letter : '\u0024' | '\u0041'..'\u005a';
ANTLR – Operators

Extended Backus-Naur Form (**EBNF**)
Demo 1
ANTLR – Troubleshooting

ANTLR does not warn about **ambiguous** rules

• resolves ambiguity at runtime
  → requires lots of testing

ANTLR does not handle indirect **left-recursion**

• direct left-recursion supported
ANTLR – Lexer ambiguity

What if some input is matched by multiple lexer rules?

```
parserRule : 'enum' parserRule;
fragment
Letter : [a-z] ;
Identifier : Letter+ ;
```

creates implicit lexer rule

```
T123 : 'enum'
```

*fragment* enforces that the rule never produces a token, but can be used in other lexer rules (e.g., `a`) can never match `enum`, but e.g., `enums`

Lexer decides based on:

1. rule with the longest match first
2. literal tokens before all regular Lexer rules
3. document order
4. *fragment* rules never match on their own
ANTLR – Parser ambiguity

```
stmt: 'if' expr 'then' stmt 'else' stmt | 'if' expr 'then' stmt | ID '=' expr ;
```

```
if a then if c then d else e
```

Ambiguous since there exist more than one parse trees for the same input.
ANTLR – Parser ambiguity

At decision points, if more than one alternative match a given input, follow **document order**.
ANTLR – Parser ambiguity

At decision points, if more than one alternative match a given input, follow **document order**.

Solution

```plaintext
stmt: 'if' expr 'then' stmt 'else' stmt
    | 'if' expr 'then' stmt
    | ID '=' expr ;
```

```plaintext
stmt: 'if' expr 'then' stmt
    | 'if' expr 'then' stmt 'else' stmt
    | ID '=' expr ;
```
ANTLR – Parser ambiguity

At **decision points**, if more than one alternative match a given input, follow **document order**.

Alternative solution:

```plaintext
stmt: 'if' expr 'then' stmt ('else' stmt)?
    | ID '=' expr ;
```

Sub-rules introduce additional decision points.
ANTLR – Left-recursion

Without: “a, b, c”

```antlr
list : LETTER (',', LETTER)*;
```

Direct:

```antlr
list : list ',' LETTER
     | LETTER ;
```

Indirect:

```antlr
list : LETTER
     | longlist ;
longlist : list ',' LETTER;
```

✓ True

✗ False
ANTLR – **Direct left-recursion**

```
exp : exp '*' exp  
    | exp '+' exp  
    | ID ;
```

A grammar that implicitly assigns **priorities** to alternatives in document order

![Rewrite examples](https://theantlrguy.atlassian.net/wiki/display/ANTLR4/Left-recursive+rules)
Demo 2
Homework

Part 2a
Parser grammar: Javali.g4

Part 2b
JavaliAstVisitor.java
Generated files

Javali**Lexer/Parser**.java
- the real thing

Javali**(Base)Visitor**.java
- base class for parse-tree visitor

Javali**(Lexer).tokens**
- token → number mapping for debugging
Generated visitor

```
start : exp EOF  
     ;
exp : exp '+' exp  
    | exp '*' exp  
    | ID  
    ;
```

one method per rule

one method per label / rule

https://theantlrguy.atlassian.net/wiki/display/ANTLR4/Parser+Rules
Constructing the Javali AST

\[
\text{start : } \text{exp} \quad \text{EOF} \\
\text{exp : } \text{exp} \quad \text{'*'} \quad \text{exp} \quad \# \text{MULT} \\
\quad | \text{exp} \quad \text{'+'} \quad \text{exp} \quad \# \text{ADD} \\
\quad | \text{ID} \quad \# \text{TERM} \\
\]

"a * a + a * a"

```
start : exp EOF 
exp : exp ' * ' exp # MULT 
    | exp ' + ' exp # ADD 
    | ID # TERM 
```

```
Var('a') 
Var('a') 
Var('a') 
Var('a')```

```
BinaryOp(*)  
BinaryOp(*)  
Var('a')  
Var('a')  
Var('a')  
Var('a')```

```
BinaryOp(+)
```

```
exp
  1
  exp
    exp
      exp
        exp
          a
            3
          a
            4
        * 
          exp
            exp
              a
                6
            * 
              exp
                exp
                  a
                    7
          + 
            exp
              exp
                a
                  5
```

```
TestBaseVisitor<T>
visitADD(ADDContext): T  
visitMULT(MULTContext): T  
visitStart(StartContext): T  
visitTERM(TERMContext): T
```
Demo 3
Notes

• You are not allowed to use syntactic predicates.
• Look on our website for more material.
• Due date is October, 22th