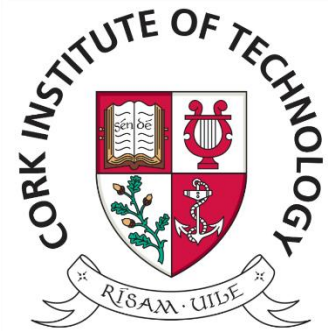
MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Programming Language Design

Lexical Analysis

Francisco Ortin

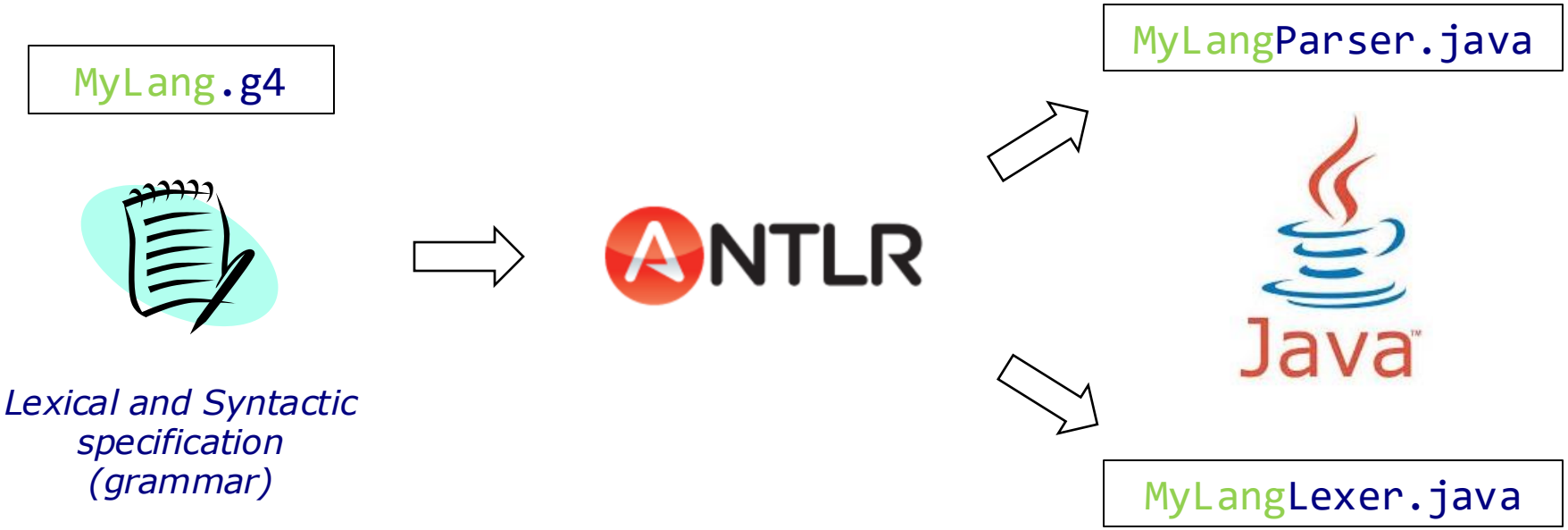


Department of
Computer Science

Contents

- Objective of the Lexical Analyzer
- Regular Expressions and Context-Free Grammars
- **Implementation of Lexical Analyzers with ANTLR**

Recall



- The interface of the **MyLangLexer** class is:
  - **nextToken**():Token The main method; each time it is called, the following token is returned

ANTLR Specification File

- The specification file has the following structure:

General Structure

Grammar Name

Options

Syntax rules

Lexical rules

Non-terminals start with lowercase

Terminals start with uppercase

Particular example (C--)

Cmm.g4

```
grammar Cmm;

@header {
   import ast.*;
   import types.*;
}

program: …
       ;
…

INT_CONSTANT: …
            ;
…
```

ANTLR Specification File

- Initially, we will just write lexical specifications (no syntax analysis yet)
- And we do not require any particular option, so the file will be

```
grammar Cmm;

program:
        ;

/* Lexical rules */

INT_CONSTANT: …
            ;
…
```

- How do we specify the lexical rules / productions?

ANTLR Specification File

- The **lexical rules** define the behavior of the lexer/scanner

  i.e., the implementation of `nextToken():Token`
- Each rule specifies the **pattern** of the different **lexemes** for a particular **token**
- Those patterns are expressed with **CFG**s in **EBNF** (Extended BNF) notation
- A very basic first example

```
grammar Cmm;

program:
        ;

INT_CONSTANT: ('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')+
            ;
```

EBNF Notation

- ANTLR patterns for **terminal** symbols (V_T)
  - Lexemes can be represented between `'` and `'`
    `'0'`, `'+'`, `'int'`
  - `\`: escape character
    - `'\''` (apostrophe), `'\\'` (backslash),
    - `\n`, `\r`, `\t`, `\b` and `\f`: special characters (`\b` = backspace, `\f` = form feed)
  - `.`: matches any character (wildcard)

- Question: Write a pattern to recognize Java / C char constants / literals

EBNF Notation

- ANTLR patterns for **terminal** symbols (V_T)
  - `'x'..'y'` (*x* and *y* being characters): matches any character between *x* and *y*, inclusively
  - `[x-y]`: identical to `'x'..'y'` (more common)
  - `[xyz]`: matches *x*, *y* or *z*; identical to `('x'|'y'|'z')` (more common)
  - `~t` (*t* being a set of characters): matches any single character not in *t*
- Question: Write a pattern to recognize a Java / C multiline comments (e.g., `/* … */`)
  - `.*? t` (*t* being a set of characters): non-greedy operator, equivalent to `(~t)* t`

EBNF Notation

- Lexemes can be represented between `'` and `'`
  `'0'`, `'+'`, `'int'`
- `\`: escape character
  - `'\''` (apostrophe), `'\\'` (backslash),
  - `\n`, `\r`, `\t`, `\b` and `\f`: special characters (`\b` = backspace, `\f` = form feed)
- `.`: matches any character (wildcard)
- `'`*x*`'..'`*y*`'` (*x* and *y* being characters): matches any character between *x* and *y*, inclusively
- `[`*x*`-`*y*`]`: identical to `'`*x*`'..'`*y*`'` (more common)
- `[`*xyz*`]`: matches *x*, *y* or *z*; identical to `('x'|'y'|'z')`
- `~`*t* (*t* being a set of characters): matches any single character not in *t*
- `.*?` *t* (*t* being a set of characters): non-greedy operator, equivalent to `(~`*t*`)*` *t*

- **Question**: Write a pattern to recognize any letter (English alphabet)

EBNF Notation

- Recall the following patterns **for any symbol** ($V_T \cup V_N$)
- Let $r,s \in (V_T \cup V_N)$
  - $r|s$: Union, matches r or s
  - $r\ s$: Concatenation, matches r and then s
  - $r*$: Kleen closure, zero or more repetitions of r
  - $r+$: Iteration, is equivalent to $rr*$
  - $r?$: Option, matches the empty input or r

Mandatory Activity

- Write an ANTLR grammar to recognize integer constants / literals
- Recall
  - Lexemes `'0'`, `'+'`, `'int'`, `'\''`
  - `.`: any character
  - `[x-y]`: *'x'..'y'*
  - `[xyz]`: `('x'|'y'|'z')`
  - `~`*t* any single character not in *t*
  - `\`: escape character
  - *r*`|`*s*: Union, matches *r* or *s*
  - *r s*: Concatenation, matches *r* and then *s*
  - *r*`*`: Kleen closure, zero or more repetitions of *r*
  - *r*`+`: Iteration, is equivalent to *rr*`*`
  - *r*`?`: Option, matches the empty input or *r*

Fragment

- It is possible to **reuse patterns**
- If a lexical pattern is too big, it is better to **break it into small patterns**
- In addition, those rules aimed at being used by other rules (i.e., they **do not define a token**) should be prefixed with the `fragment` keyword

```
grammar Fragment;

program: ;

fragment
DIGIT: [0-9]
     ;

INT_CONSTANT: '0'
            | [1-9] DIGIT*
            ;
```

Skip

- As mentioned, one of the objectives of the lexer is to discard meaningless characters (e.g., new line, tabs, comments…)
- ANTLR provides this functionality with **lexical rules that specify the lexemes to be discarded**, adding `-> skip` at the end of the production

```
grammar Skip;

program: ;

WHITE_SPACES: ' '+ -> skip
            ;
```

nextToken():Token

- So, what happens if?
  - No pattern is matched?
  - Two patterns are matched?
- What is the **algorithm** of the generated `nextToken()`?

```
Token nextToken() {
  while(current character is not end-of-file) {
    if (any pattern matches)
      return the token matching the first pattern
           that recognizes the longest lexeme
    else {
      System.err.println("line x:y token  " +
            "recognition error at 'character'");
      ignore character
  }   }
  return new Token(MyLangParser.EOF);
}
```

Mandatory Activity

- The following scanner recognizes integer literals

```
grammar IntLiteralsLang;

program:
        ;

INT_CONSTANT: '0'
            | [1-9][0-9]*
            ;
```

- What happens if a space, tabulation, line feed or carriage return appears?
- How can we solve it?
- Which tokens are recognized for the following input? `129 0102`

Mandatory Activity

- What does the following scanner return for the following source programs?

Source programs:

int	while	variable	integer	hi	int3

KewordsAndIDsLang.g4

```
grammar KewordsAndIDsLang;

program: ;

INT: 'int' ;
WHILE: 'while' ;
ID: [a-z]+ ;
WS: [ \t\n\r]+ -> skip ;
```

Autonomous Activity

- Write an **ANTLR lexical specification** file for the following patterns:
  - Identifiers

    `var1`, `a`, `var_2`, `__private`, `_`
  - Real constants (without exponent)

    `0.0`, `1.`, `.45`
  - Ignore single line comments

    `// This is one single-line comment`

Bibliography

- Alfred V. Aho, Monica S. Lam. Compilers: Principles, Techniques, and Tools, 2 Edition. Addison Wesley, 2006.
- Terence Parr. The Definitive ANTLR 4 Reference, 2nd edition. Pragmatic Bookshelf, 2013.
- Kenneth C. Louden. Compiler Construction, Principles and Practice. PWS Publishing, 1997.