Compiler Design

Lecture 3: Introduction to Lexical Analysis

Christophe Dubach

Winter 2022

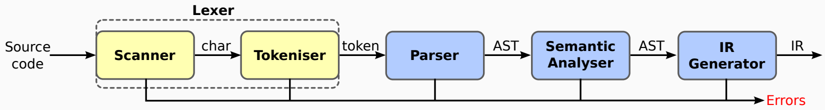
Timestamp: 2022/01/05 15:05:58

Reminder

Action

Fill up online form with your name and userid
(instructions online on gitlab)

The Lexer



- Maps character stream into words — the basic unit of syntax
- Assign a syntactic category to each work (part of speech)
  - $x = x + y$; becomes ID(x) EQ ID(x) PLUS ID(y) SC
  - word $\cong$ lexeme
  - syntactic category $\cong$ part of speech
  - In casual speech, we call the pair a token
- Typical tokens: number, identifier, `+`, `-`, `new`, `while`, `if`, ...
- Scanner eliminates white space (including comments)

Table of contents

Languages and Syntax

Languages and Syntax

Context-free Language

Context-free Language

Context-free syntax is specified with a grammar.

For instance:

```
SheepNoise → SheepNoise baa
           | baa
```

This grammar defines the set of noises that a sheep makes (under normal circumstances).

It is written in a variant of Backus–Naur Form (BNF).

Formally

G $=$ (S,N,T,P) is a grammar where

- S is the start symbol
- N is a set of non-terminal symbols
- T is a set of terminal symbols or words
- P is a set of productions or rewrite rules (P:N $\rightarrow$ N $\cup$ T)

Example

```
1   goal  →  expr
2   expr  →  expr op term
3         |  term
4   term  →  number
5         |  id
6   op    →  +
7         |  −
```

$$
\begin{aligned}
S &= \text{goal} \\
T &= \{\text{number}, \text{id}, +, -\} \\
N &= \{\text{goal}, \text{expr}, \text{term}, \text{op}\} \\
P &= \{1, 2, 3, 4, 5, 6, 7\}
\end{aligned}
$$

This grammar defines simple expressions with addition & subtraction over “number” and “id”.

This grammar, like many, falls in a class called context-free grammars, abbreviated CFG

- context-free ⇒ the left hand-side only contains a single non-terminal symbol

Empty symbol ϵ

A grammar can also contain a special empty symbol ϵ

For instance:

```
goal → A | ε
A    → A
     | a
```

Recognizes the following set of inputs: $\{\epsilon, a, aa, aaa, \ldots\}$ where ϵ represents an empty input.

Languages and Syntax

Regular Expression

Regular Expression

Grammars can often be simplified and shortened using an augmented BNF notation where:

- $x*$ is the Kleene closure : zero or more occurrences of x
- $x+$ is the positive closure : one or more occurrences of x
- $[x]$ is an option: zero or one occurrence of x

Example: identifier syntax

```
identifier ::= letter (letter | digit)*
digit      ::= "0" | ... | "9"
letter     ::= "a" | ... | "z" | "A" | ...| "Z"
```

Exercise: write the grammar of signed natural number

Languages and Syntax

Regular Languages

Regular Language

Definition

A language is regular if it can be expressed with a single regular expression or with multiple non-recursive regular expressions.

- Regular languages can be used to specify the words to be translated to tokens by the lexer.
- Regular languages can be recognised with finite state machine.
- Using results from automata theory and theory of algorithms, we can automatically build recognisers from regular expressions.

Regular language to program

Given the following:

- c is a lookahead character;
- next() consumes the next character;
- error () quits with an error message; and
- first (exp) is the set of initial characters of exp.

Then we can build a program to recognise a regular language.

RE	pr(RE)
“x”	if (c == 'x') next() else error ();
(exp)	pr(exp);
$[exp]$	if (c in first (exp)) pr(exp);
$exp*$	while (c in first (exp)) pr(exp);
$exp+$	pr(exp); while (c in first (exp)) pr(exp);
$fact_1 \ldots fact_n$	pr(fact1); ... ; pr(factn);

$term_1 \mid \ldots \mid term_n$	switch (c) {
	  case c in first(term1) : pr(term1);
	  case ... : ... ;
	  case c in first(termn) : pr(termn);
	  default : error();
	}

This only works if the grammar is left-parsable.

Definition: left-parsable

A grammar is left-parsable if:

$term_1 \| \ldots \| term_n$	The terms do not share any initial symbols.
$fact_1 \ldots fact_n$	If $fact_i$ contains the empty symbol then $fact_i$ and $fact_{i+1}$ do not share any common initial symbols.
$[exp], exp*$	The initial symbols of *exp* cannot contain a symbol which belong to the first set of an expression following *exp*.

Left-parsable grammar examples

```
G  ::= A | B
A  ::= 'a' 'b' // first(A) = {'a'}
B  ::= 'c'     // first(B) = {'c'}
```

input : "ab"

```
G  ::= [A] B
A  ::= 'a' | 'b' // first(A) = {'a','b'}
B  ::= 'c'       // first(B) = {'c'}
```

input : "bc"

Non left-parsable grammar examples

```
G   ::= A | B
A   ::= 'a' 'b' // first(A) = {'a'}
B   ::= 'a' 'c' // first(B) = {'a'}
```

input : "ac"

```
G   ::= [A] B
A   ::= 'a' | 'b' // first(A) = {'a','b'}
B   ::= 'b' 'c'   // first(B) = {'b'}
```

input : "bc"

```
G   ::= A B
A   ::= 'a' | 'b' | ϵ // first(A) = {'a', 'b', ϵ}
B   ::= 'b' 'c'       // first(B) = {'a'}
```

input : "bc"

Example: recognizing identifiers

Identifier syntax (example)

```
identifier ::= letter (letter | digit)*
digit      ::= "0" | ... | "9"
letter     ::= "a" | ... | "z" | "A" | ...| "Z"
```

Java-ish Program

```
void ident() {
  if (c is in [a-zA-Z])
    letter();
  else
    error();
  while (c is in [a-zA-Z0-9]) {
    switch (c) {
      case c is in [a-zA-Z] : letter();
      case c is in [0-9] : digit();
      default : error();
    }
  }
}
void letter() {...}
void digit() {...}
```

More “realistic” Java version

```java
void ident() {
  if (Character.isLetter(c))
    next();
  else
    error();
  while (Character.isLetterOrDigit(c))
    next();
}
```

Lexical Analysis

Lexical Analysis

Building a Lexer

Role of lexical analysiser

The main role of the lexical analyser (or lexer) is to read a bit of the input and return a lexeme (or token).

Java Lexer class:

```
class Lexer {
  public Token nextToken() {
    // return the next token, ignoring white spaces
  }
  ...
}
```

White spaces are usually ignored by the lexer. White spaces are:

- white characters (tabulation, newline, . . .)
- comments (any character following “//” or enclosed between “/*” and “*/”

What is a token?

A token consists of a token class and other additional information.

Example: some token classes

IDENTIFIER	→	foo, main, cnt, ...
NUMBER	→	0, −12, 1000, ...
STRING_LITERAL	→	"Hello world!", "a", ...
EQ	→	==
ASSIGN	→	=
PLUS	→	+
LPAR	→	(
...	→	...

Java Token class:

```
class Token {
  TokenClass tokenClass; // Java enumeration
  String data;           // stores number or string
  Position pos;          // line/column number in source
}
```

Example

Given the following C program:

```
int foo(int i) {
  return i+2;
}
```

the lexer will return:

```
INT IDENTIFIER("foo") LPAR INT IDENTIFIER("i") RPAR LBRA
  RETURN IDENTIFIER("i") PLUS NUMBER("2") SEMICOLON
RBRA
```

A Lexer for Simple Arithmetic Expressions

Example: BNF syntax

```
identifier ::= letter (letter | digit)*
digit      ::= "0" | ... | "9"
letter     ::= "a" | ... | "z" | "A" | ... | "Z"
number     ::= digit+
plus       :: = "+"
minus      :: = "-"
```

Example: token definition

```
class Token {

    enum TokenClass {
        IDENTIFIER
        NUMBER,
        PLUS,
        MINUS
    }

    // fields
    final TokenClass tokenClass;
    final String data;
    final Position position;

    // constructors
    Token(TokenClass tc) {...}
    Token(TokenClass tc, String data) {...}
    ...
}
```

Example: tokeniser implementation

```
class Tokeniser {

  Scanner scanner;

  Token next() {
    char c = scanner.next();

    // skip white spaces
    if (Character.isWhitespace(c)) return next();

    if (c == '+') return new Token(TokenClass.PLUS);
    if (c == '-') return new Token(TokenClass.MINUS);

    // identifier
    if (Character.isLetter(c)) {
      StringBuilder sb = new StringBuilder();
      sb.append(c);
      c = scanner.peek();
      while (Character.isLetterOrDigit(c)) {
        sb.append(c);
        scanner.next();
        c = scanner.peek();
      }
      return new Token(TokenClass.IDENTIFIER, sb.toString());
    }
```

Example: continued

```
    // number
    if (Character.isDigit(c)) {
      StringBuilder sb = new StringBuilder();
      sb.append(c);
      c = scanner.peek();
      while (Character.isDigit(c)) {
        sb.append(c);
        scanner.next();
        c = scanner.peek();
      }
      return new Token(TokenClass.NUMBER, sb.toString());
    }

  }

}
```

Lexical Analysis

Ambiguous Grammar

Some grammars are ambiguous.

Example 1

```
comment ::= "/*" .* "*/" | "//" .* NEWLINE
div     ::= "/"
```

Solution:

Longest matching rule

The lexer should produce the longest lexeme that corresponds to the definition.

Project hint: comments are actually considered a special case. Use peek ahead function from the Scanner, and assume that /* and // always indicate the start of a comment.

Some grammars are ambiguous.

Example 2

```
number      ::=  ["-"] digit+
digit       ::=  "0" | ... | "9"
plus        :: = "+"
minus       :: = "-"
```

Exampple input: -9

`number` or `minus` `number` ?

Solution:

Delay to parsing stage

Remove the ambiguity and deal with it during parsing

```
number      ::=  digit+
digit       ::=  "0" | ... | "9"
plus        :: = "+"
minus       :: = "-"
```

Next lecture

- Automatic Lexer Generation