Compiler Design

Lecture 3: Introduction to Lexical Analysis

Christophe Dubach

Winter 2024

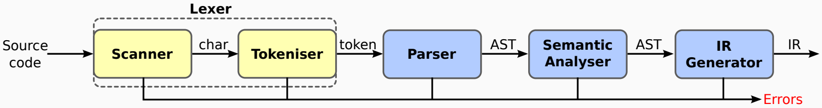
Timestamp: 2024/01/09 15:25:00

Reminder

Action

Fill up online form with your name and userid
(link available on course webpage)

The Lexer



The Lexer:

- Produces a stream of characters from the source code;
- Separates the stream into lexems — the basic unit of syntax
  - A lexem is similar to a “word” in natural languages
- and assigns a syntactic category to each lexem (part of speech)
  - For natural languages : noun, verb, adjective, ...
  - For programming languages : number, keyword, idenfifier, `+`, `(`, ...
- to produce a sequence of tokens (pair of lexem + category)

For instance, `x = x+y;` is turned by the lexer into:
`ID(x) EQ ID(x) PLUS ID(y) SC`

Note that the lexer eliminates white spaces (including comments).

Table of contents

Languages and Syntax

- Context-free Language
- Regular Expression
- Regular Languages

Lexical Analysis

- Building a Lexer
- Ambiguous Grammar

Languages and Syntax

Languages and Syntax

Context-free Language

Context-free Language

Context-free syntax is specified with a context-free 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 ($P{:}N \rightarrow N \cup T$)

A context-free grammar, abbreviated CFG, is a grammar where the left hand-side of each production rule only contains a single non-terminal symbol.

Example of context-free grammar

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

```
S = goal
T = {number, id , + , -}
N = {goal , expr , term , op}
P = {1 ,2 ,3 ,4 ,5 ,6 ,7}
```

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

Only non-terminal symbols appear on the left hand-side of the rules.

It means we can always produce an expression by subtituting the left hand-side with any of the choices on the righ hand-side. For instance:

goal → expr → expr op term → term op term → number + id

Example of non-context-free grammar:

```
A    →  B
B    →  b  B
      |  C
b  C →  c
```

Let’s try to derive some expressions with this grammar:

- A → B → b B → b b B → b b C → b c
- A → B → C → ???

The application of the last rule depends on context.

This means we need to keep track of what has happened in the past (and we can get stuck) ⇒ harder!

Empty symbol ϵ

A grammar can also contain a special empty symbol ϵ

For instance:

```
goal → A | ϵ
A    → Aa
       | 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 *lexem* to be translated to tokens by the lexer.

Biggest advantage: a regular language can be recognised with a finite state machine.

Using results from automata theory and theory of algorithms, we can automatically build recognisers from regular expressions (topic of next lecture).

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 );`

$term_1 | \ldots | term_n$

```
switch (c) {
  case c in first(term1) : pr(term1);
  case ...               : ...      ;
  case c in first(termn) : pr(termn);
  default : error();
}
```

RE = Regular Expression, pr = program

This only works if the grammar is left-parsable.

Definition: left-parsable

A grammar is left-parsable if:

$term_1 \| \dots \| term_n$	The terms do not share any initial symbols.
$fact_1 \dots fact_n$	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) = {'b'}
```

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() {
  if (c is in [a-zA-Z]) next();
  else error();
}
void digit() {
   if (c is in [0-9]) next();
   else error();
}
```

More “realistic” Java version

```
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 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 category and other additional information.

Example of token categories

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

Java Token class:

```
class Token {
  Category category; // 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 Category {
        IDENTIFIER
        NUMBER,
        PLUS,
        MINUS,
        INVALID
    }

    // fields
    Category category;
    String data;
    Position position;

    // constructors
    Token(Category cat) {...}
    Token(Category cat, 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(Category.PLUS);
    if (c == '-') return new Token(Category.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(Category.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(Category.NUMBER, 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(Category.NUMBER, sb.toString());
    }

    // else
    error();
    return new Token(Category.INVALID);
  }

}
```

Lexical Analysis

Ambiguous Grammar

Some grammars are ambiguous.

Example 1

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

Solution:

Longest matching rule

The lexer should recognized 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       :: = "-"
```

Example input: `-9`

Is it `number` or `minus` `number` ?

Some grammars are ambiguous.

Example 2

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

Example input: `-9`

Is it `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