1 Part II: Lexical Analysis (Scanner)

- A scanner is described by a *regular* language.
- Handwritten scanners.
- Generated scanners (by JLex).

2 Regular Languages

- A language is *regular* if its syntax can be expressed by a single EBNF rule without recursion.
- Since there is only one, non-recursive rule all symbols on the right-hand side must be terminal symbols. The right-hand side is also called a *regular expression.*
- Regular languages are interesting because they can be recognized by *finite state machines.*
- Alternatively, a language is regular if its syntax can be described by a number of EBNF rules without recursion.

Example:

```
NUMBER ::= DIGIT { DIGIT }.
DIGIT  ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```


3 Lexical Analysis / Syntactic Analysis

- The syntax of a programming language is given in two stages.
- *Lexical Syntax* describes the form of individual tokens (words).
- *Context-free Syntax* describes how programs are formed out of tokens.
- The translation of source programs into token sequences is the main task of the *lexical analyzer (scanner)* component in a compiler.
- *Lexical Syntax* is usually described by a *regular language*
- Hence, lexical analyzers can be implemented by finite state machines.
- For the *Context-free Syntax* finite state machines are not powerful enough. Programming languages are usually not regular.

4 Exercise

- We have a variable ch, which contains the current character. This variable is called *lookahead.*
- We have function **int** nextCh() which reads the next input character. It is used to set ch with ch = nextCh()
- We have a function **void** error() which quits with an error message.

Write a function **void** readBinNumber() which reads the next binary number.

```
BINNUMBER ::= BINDIGIT { BINDIGIT }.
BINDIGIT  ::= "0" | "1".
```

At the beginning the first character is already in ch. When the function returns, the first character after the binary number should be in ch.

If the input does not start with a binary number it should call error().

5 From a Regular Language to Program Code

Expr	Prog(Expr)
"x"	**if** (ch == 'x') { ch = nextCh(); } **else** { error(); }
(E)	Prog(E)
$[E]$	**if** (ch in first(E)) { Prog(E) }
$\{E\}$	**while** (ch in first(E)) { Prog(E) }
EF	Prog(E) Prog(F)
$E \mid F$	**if** (ch in first(E)) { Prog(E) } **else** { Prog(F) }

first(E) is the set of terminals, E can start with.

Q: What is first(BINNUMBER)?

If we use multiple rules, each rule gives one procedure.

6 Straightforward Generation

```
void readBinNumber() {
        readBinDigit();
        while (ch == '0' || ch == '1') {
                readBinDigit();
        }
}

void readBinDigit() {
        if (ch == '0') {
                if (ch == '0') { ch = nextCh(); } else { error(); }
        } else {
                if (ch == '1') { ch = nextCh(); } else { error(); }
        }
}
```


7 Optimized Version

```
void readBinNumber() {
        if (ch == '0' || ch == '1') { ch = nextCh(); } else { error(); }
        while (ch == '0' || ch == '1') {
                ch = nextCh();
        }
}
```

- Use inlining.
- Leave out unnecessary ifs.
- Replace if-then-else chains by switches
- Remove ifs and switches, when the alternatives do the same thing

8 Possible Problems

The method, however, does not always work:

- ONEORTWO ::= [BINDIGIT] BINDIGIT

```
void readOneOrTwo() {
        if (ch == '0' || ch == '1') { ch = nextCh(); }
        if (ch == '0' || ch == '1') { ch = nextCh(); } else { error(); }
}
```

- ONEORMORE ::= { BINDIGIT } BINDIGIT
- INTORFLOAT ::= NUMBER | NUMBER "." NUMBER

Q: Can you find equivalent expressions, that do not have the problem?

- These problems can always be resolved for regular expressions.
- We cannot solve them in general, if the grammar has recursion.

9 The Task of the Lexical Analyzer

- So far, we checked one kind of token (binary numbers).
- Usually, a scanner has to recognize a variety of tokens and to return the one it found.
- A scanner also has to skip white space and comments.
- For some tokens the scanner needs to collect additional information:
  - Which number was it?
  - The source position of the character.

10 Examples

Example 1:

3 ∗ (5 + 3) /∗ small comment ∗/ − 7

The Scanner should give:

INTLIT(3), TIMES, LPAREN, INTLIT(5), PLUS,
INTLIT(3), RPAREN, MINUS, INTLIT(7), EOF

Example 2:

3 ∗ + &

The Scanner should give: INTLIT(3), TIMES, PLUS, ERROR, EOF

11 A Handwritten Lexical Analyzer

- We write a function nextToken() which reads the next token and returns a different integer for each different kind of token.
- The basic principle is the same ch, nextCh()
- Errors are sometimes delegated to the next phase by returning a special ERROR token.
- If there is no more input it returns a special EOF token.
- Sometimes we need to return more information (which number for integer literals)
- The function nextToken() stores this in a predefined variable.
- Alternatively, it can return a token object, that contains the token number and the additional information.

12 A Handwritten Lexical Analyzer

```
Object obj;                // additional information on token
int pos;                   // position of token in source
int nextToken() {
    while (ch == ' ' || ch == '\t'
              || ch == '\n || ch == '\r) { ch = nextCh(); }
    pos = ...;             // set position
    switch (ch) {
        case '+': { ch = nextCh(); return PLUS; }
        case '0': case '1': ... case '9': {
            ...;      // scanning integers
            obj = new Integer(...); return INTLIT;
        }
        default: { ch = nextCh(); return ERROR; }
    }
}
```

13 The Longest Match Rule

When does one token end and the next token start?

- Q: what do the following java expressions mean?
  Are they valid?
  (x +++ y), (x + ++ y), (x ++++ y)
- Solution: The scanner matches at each step the *longest* possible token.

  - The first is (x ++ + y), add then increment x.
  - The second is (x + ++ y), increment y then add.
  - The third is (x ++ ++ y), which is invalid.
- We have already done this when reading binary numbers.

14 Using a Scanner Generator

- The input consists of a list of pairs (*pattern*, *action*).

[0–9]+	{ **return** mkToken(INTLIT, input); }
"+"	{ **return** mkToken(PLUS); }
"–"	{ **return** mkToken(MINUS); }
" "	{ ; }

- The scanner generator uses this input to build a source file for the actual scanner.
- In our project we will use the scanner generator JLex
  http://www.cs.princeton.edu/~appel/modern/java/JLex/
- Whenever a pattern is recognized in the input, the action is performed.
- The patterns are regular expressions (For JLex we have to use a different syntax).
- Typically the action returns the token.
- The longest possible input string is matched.
- If multiple input strings match, the action of the earlier is performed.

15 Regular Expressions in JLex

- Apart from the special characters
  ? * + | ( ) ^ $ / ; . = < > [ ] { } " \ and blank, every character stands for itself.
  (Example: **while**)
- | and () have the same meaning as in EBNF. (Example: a(b|c)
- E* is the same as EBNF { E }. ab* is the same as EBNF "a" { "b" }.
- E? is the same as EBNF [E].
- E+ is the same as EBNF E { E }.
- After \ the special characters lose their special meaning.
  (Example: \+)
- Between double quotes " all special characters but \ and " lose their special meaning. (Example: "+")
- The following escape sequences are recognized: \b \n \t \f \r.

16 Regular Expressions in JLex (2)

- name = E is used to define macros. (Example: PLUS = (\+))
- {name} refers to the macro name (Example: {PLUS})
- With [ ] we can describe sets of characters.

  - [abc] is the same as EBNF ("a" | "b" | "c")
  - [a–d] is the same as EBNF ("a" | "b" | "c" | "d")
- With [^] we can describe sets of characters.

  - [^\n\"] means anything but a newline or quotes
  - [^a–z] means anything but a lower-case letter
- We can use . as a shortcut for [^\n]

17 Examples

- [0–9]+ describes integer numbers.
- \"[^\"\n]*\" describes simple strings without newlines (verb@ is not treated specially)..

Q: Write JLex regular expressions for

- integer numbers which do not have superfluous leading zeroes
- ...

18 JLex Example: Expressions

```
package expression;
import java_cup.runtime.*;

%%

%cup
%class Scanner

%eofval{
    return mkToken(Tokens.EOF);
%eofval}

%{
    // arbitrary Java code
    // code for position and debugging
%}
```



```
%line
%char

DIGIT     = [0–9]
WHITE     =

%%

"+"       { return mkToken(Tokens.PLUS); }
"–"       { return mkToken(Tokens.MINUS); }
"*"       { return mkToken(Tokens.TIMES); }
"/"       { return mkToken(Tokens.DIV); }
"("       { return mkToken(Tokens.LPAREN); }
")"       { return mkToken(Tokens.RPAREN); }

{DIGIT}+  { return mkToken(Tokens.INTLIT, new Integer(yytext())); }

{WHITE}   { /* ignore white space. */ }
.         { Report.error(position(), "Illegal character: " + yytext()); }
```


19 Tokens

This class will later be generated by the parser generator.

```
package expression;
interface Tokens {
    public static final int EOF    = 0;
    public static final int PLUS   = 1;
    public static final int MINUS  = 2;
    public static final int TIMES  = 3;
    public static final int DIV    = 4;
    public static final int LPAREN = 5;
    public static final int RPAREN = 6;
    public static final int INTLIT = 7;
}
```

20 java_cup.runtime.Symbol

To be able to use the scanner with the parser we use later, we have to put the result in a format that the parser understands. This class comes with the parser generator.

```
package java_cup.runtime;
public class Symbol
{
        public int sym;
        public int left, right;
        public Object value;
        ...
        public Symbol(int sym, int left, int right, Object value) {
                ...
        }
}
```

Our function mkToken constructs objects of that kind.

21 ScannerTest

```
package expression;
import java_cup.runtime.*;

class ScannerTest {
    public static void main(String args[ ]) throws java.io.IOException {
        Scanner scanner = new Scanner(System.in);   // scan from stdin
        java_cup.runtime.Symbol sym;
        do {
            sym = scanner.next_token();   // read one sym from scanner
            System.out.println(Scanner.toString(sym));   // print it
        } while (sym.sym != Tokens.EOF);   // until EOF reached
    }
}
```