1 Part III: Lexical Analysis (Scanner)

- Lexical analysis / syntactic analysis.
- 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.
- *Micro-Syntax* describes the form of individual tokens (words).
- *Macro-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* component in a compiler.
- *Micro-Syntax* is usually described by a *regular language*
- Hence, lexical analyzers can be finite state machines.
- For the *Macro-Syntax* finite state machines are not powerful enough. Programming languages are usually not regular.

4 Exercise

Assume you have

- a variable ch, which contains the current character. This variable is called *lookahead*.
- a function **void** nextCh() which sets the variable ch to the next input character.
- a function error() which quits with an error message.

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

```
BinNumber = BinDigit { BinDigit }.
BinDigit  = "0" | "1".
```

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

Did the grammar for numbers help you in writing the function?

5 From a Regular Language to Program Code

Expr	Prog(Expr)
"x"	**if** (ch == 'x') 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') nextCh(); else error();
    } else {
        if (ch == '1') nextCh(); else error();
    }
}
```


7 Optimized Version

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

This, however, does not always work:

- [BinDigit] BinDigit

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

- { BinDigit } BinDigit
- Number | Number "." Number

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

- These problem 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

- The basic action of a lexical analyzer is to read some part of the input and to return a token at each call.
  If there is no more input it returns a special EOF-token.

```
Token token;    /* last token read */
Object obj;     /* additional information on token */
int pos;        /* position of token in source */
void nextToken() {
        /* skip white space and comments, assign the next
           token to token and additional information to obj.
           set pos to the position of the token */
}
```

  In JLex nextToken will actually return an object of class Symbol containing token, obj, pos.
- Throw away *white space* and *comments* in between tokens.
- When does one token end and the next token start?

10 White Space and Comments

- White space has to be skipped:
  - blank character, tabulator, newline
  - more general: any character (**char**)0 $\leq$ c $\leq$ ' '.
- Comments as well:
  - /*...*/ where ... does not contain */.
  - from // to the next end-of-line.

11 The Longest Match Rule

- 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.

12 Example

3 * (5 + 3) /* small comment */ – 7

The Scanner should give:

numlit(3), times, lparen, numlit(5), plus,
numlit(3), rparen, minus, numlit(7), eof

13 Using a Scanner Generator

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

digit { digit }	**return** mkToken(number, input);
"+"	**return** mkToken(plus);
"−"	**return** mkToken(minus);
" "	;

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

14 Regular Expressions in JLex

- Apart from the special characters ? * + | ( ) ^ $ / ; . = < > [ ] { } " \ and blank, every character stands for itself.
- After \ the special characters lose their special meaning.
- Between double quotes " all special characters but \ " lose their special meaning.
- The following escape sequences are recognized: \b \n \t \f \r \ddd \xdd \udddd \^C \c.
- | and () have the same meaning as in EBNF.
- {name} refers to the non-terminal name
- name = E is used to define non-terminals.
- 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 }.

15 Regular Expressions in JLex (2)

- [S] means any of a set of characters described by S. [^S] means any character not described by S.
  - [abc] is the same as EBNF ("a" | "b" | "c")
  - [a–cxyz] is the same as EBNF ("a" | "b" | "c" | "x" | "y" | "z")
  - [^xz] matches anything but x or z
- . matches everything but a newline. It is the same as [^\n]

Examples:

- [0–9]+ describes integer numbers.
- \"[^\"]*\" describes simple strings without escapes.

Q: Write JLex regular expressions for Number, noStarOrSlash and floating point numbers!

16 JLex Example: Expressions

```
package expression;
import jaco.framework.parser.Symbol;

%%

%class Scanner
%implements Tokens
%function nextToken
%type Symbol

%eofval{
    return mkToken(EOF);
%eofval}
```

```
%{
    static String representation(int token) {
        switch(token) {
            case PLUS: return ("+");
            case MINUS: return ("-");
            case TIMES: return ("*");
            case DIV: return ("/");
            case LPAREN: return ("(");
            case RPAREN: return (")");
            case NUMLIT: return ("numlit");
            case EOF: return ("EOF");
            default: return ("<unknown>");
        }
    }
```



```
        protected int position() {
            return Position.encode(yyline + 1,
                yychar - firstCharInLine);
        }
        protected int rposition() {
            return Position.encode(yyline + 1,
                yychar - firstCharInLine + yylength() - 1);
        }
        protected int firstCharInLine = -1;
        protected Symbol mkToken(int token) {
            return new Symbol(token, position(), rposition());
        }
        protected Symbol mkToken(int token, Object obj) {
            return new Symbol(token, position(), rposition(), obj);
        }
    %}
```



```
%line
%char

LETTER      = [A-Za-z_]
DIGIT       = [0-9]

%%

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

{DIGIT}+    { return mkToken(NUMLIT, new Integer(yytext())); }
[\ \t]      { }
[\n\r]      { firstCharInLine = yychar; }
.           { Report.error(position(), "Illegal character: " + yytext()); }
```


17 Tokens

This class will later be generated by the parser generator.

```
package expression;
interface Tokens {
    int EOF     = 0;
    int PLUS    = EOF + 1;
    int MINUS   = PLUS + 1;
    int TIMES   = MINUS + 1;
    int DIV     = TIMES + 1;
    int LPAREN  = DIV + 1;
    int RPAREN  = LPAREN + 1;
    int NUMLIT  = RPAREN + 1;
}
```

18 Symbol

This class comes with the parser generator.

```
package jaco.framework.parser;
public class Symbol
{
    public int sym;
    public int left, right;
    public Object value;
    ...
    public Symbol(int sym, int left, int right, Object value) {
        ...
    }
}
```


19 ScannerTest

```
package expression;
class ScannerTest {
    public static void main(String args[ ]) throws Exception {
        Scanner scanner = new Scanner(System.in);
        jaco.framework.parser.Symbol sym;
        do {
            sym = scanner.nextToken();
            System.out.println("Token("
                + Position.posToString(sym.left) + ", "
                + Position.posToString(sym.right) + "): "
                + scanner.representation(sym.sym)
                + ((sym.value != null)
                    ? "(" + sym.value + ")" : ""));
        } while (sym.sym != Tokens.EOF);
    }
}
```


20 How does generation work?

- There is a systematic way to map any regular expression to a lexical analyzer
- Three steps
  - regular expression ⇒nondeterministic finite state automaton
  - nondeterministic finite state automaton ⇒deterministic finite state automaton
  - deterministic finite state automaton ⇒scanner program

21 Finite State Automata

- Consist of a finite number of *states* and *transitions*
- Transitions are labelled with input characters.
- There is one *start state*.
- A subset of the states are called *final states*.
- A finite state automaton starts in the start state, and for each input symbol follows an edge labelled with that symbol.
- It *accepts* an input string iff it ends up in a final state.
- Examples: Blackboard.

22 (Non)Deterministic Finite state Automata

- In a *nondeterministic finite state automaton* (NFA), there can be more than one edge originating from the same node and labelled with the same label.
- Or there can be a special ϵ-edge, which can be followed without consuming any input symbols.
- By contrast, in a *deterministic finite state automaton* all edges leaving some node have pairwise disjoint label sets and there are no ϵ-labels.

23 From a Regular Expression to an NFA

24 From an NFA to a DFA

- Problem: Executing an NFA needs *backtracking*, which is inefficient.
- We would prefer a DFA.
- Idea: Do all possible choices in parallel.
- Construct a DFA, which has a state for each possible set of NFA states.
- A DFA state is final if the set of its NFA states contains a final state.
- Since the number of states of an NFA is finite (say N), the number of possible sets of states is also finite (bounded by 2^N).
- Often the number of reachable sets of states is much smaller.

25 Algorithm

- First step: For a set of states S, let closure(S) be the largest set of states, that is reachable from S using only ϵ-transitions.
- Algorithm to compute T = closure(S):

```
T = S;
do {
    T' = T;
    for each state s ∈ T {
        for each edge e from s to some s' {
            if (e is labelled with ε) {
                T = T ∪ s';
            }
        }
    }
} while (T != T')
```

- This is an example of a fixpoint algorithm.

26 Algorithm (2)

- Second step: For a set of states S and an input symbol c, let DFAedge(S,c) be the set of states that can be reached from S by following an edge labelled with c.
- Algorithm to compute T=DFAedge(S,c)

```
T = ∅;
for each state s ∈ S {
    for each edge e from s to some s' {
        if (e is labelled with c) {
            T = T ∪ closure({s'});
        }
    }
}
```


27 Simulating a DFA

- Using the machinery developed so far, we can already *simulate* a DFA, given an NFA.
- Let s be the start state. Then the simulation works as follows

```
d = closure({s});
while (ch != EOF) {
    d = DFAedge(d, ch);
    nextCh();
}
```

- Manipulating these sets at runtime is still very inefficient.

28 DFA Construction

- DFA-states are numbered from 0.
- 0 is the error state, corresponding to the empty set of NFA-states. The DFA goes into state 0, iff the NFA would have blocked because no edge matched the input symbol.
- states is an array which maps each DFA-state to the set of NFA states it represents. trans is a matrix of transitions from state numbers to state numbers.

29 DFA Construction (2)

- Algorithm:

```
states[0] = ∅; states[1] = closure({s});
j = 0; p = 2;    /* states[0..j–1] done, state[j..p–1] to do */
while (j < p) {
    for each input character c {
        d = DFAedge (states[j], c);
        if (d == states[i] for some i < p)
            trans[j, c] = i;
        else {
            states[p] = d;
            trans[j, c] = p;
            p = p + 1;
        }
    }
    j = j + 1;
}
```

30 Executing a DFA

- use trans

```
s = 1;
while (ch != EOF) {
        s = trans[s, ch];
        nextCh();
}
```


31 Executing a DFA

- generate switch

```
s = 1;
while (ch != EOF) {
        switch (s) {
        case 0: error(); break;
        case 1:
                switch (ch) {
                case 'a': s = 3; break;
                ...
                }
                break;
        ...
        }
        nextCh();
}
```


32 Summary: Lexical Analysis

- Lexical analysis turns input characters into tokens.
- Lexical syntax is described by regular expressions.
- We have learned two ways to construct a lexical analyzer from a grammar for lexical syntax.
- By hand, using a program scheme
- By machine, using JLex to construct of DFA.
- Scanner generator / hand-written scanner
  - Speed
  - Size
  - Flexibility
  - Maintenance
  - Ease of Coding

33 Exercise

Consider "a" { ("b" | { "a" } "c") "x" } | { "x" } "a".

- Find an NFA for this expression!
- Convert this NFA into a DFA!