Part II : Lexical Analysis

- Regular Languages
- Translation from regular languages to program code
- A grammar for JO
- Context-free Grammar of JO
- Assignment 1

Regular Languages

Definition : *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 of the production must be terminal symbols. The right-hand side is also called a *regular expression.*

Regular languages are interesting since they can be recognised by *finite-state machines.*

Alternatively, a language is regular if its syntax can be expressed by a number of EBNF rules, but no recursion between the rules is allowed.

Example :

```
identifier = letter {letter | digit}
digit = "0" | ... | "9 »
letter = "a" | ... | "z" | "A" | ... | "Z"
```

Regular Languages and Lexical Analysis

- The syntax of a programming language is usually given in two stages.
- *Micro-Syntax* describes the form of individual words or tokens.
- *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.
- What kind of programs correspond to finite state machines?

Exercise

Assume you have a function

```
char next ();
```

which returns the next input character.

Write a function

```
boolean isIdent ()
```

which tests whether the input is of the form

input = identifiier '\n'.

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

In what way ?

Translation from regular languages to program code

K	Pr(K)
"x"	**if** (sym == "x") next(); **else** error();
(exp)	Pr(exp)
[exp]	**if** (« sym in first(exp) ») { Pr(exp) }
{exp}	**while** (« sym in first(exp) ») { Pr(exp) }
$fact_1$ ... $fact_n$	Pr($fact_1$) ; ... ; Pr($fact_n$)

$term_1$ | ... | $term_n$	**switch** (sym) {
	case first($term_1$): Pr($term_1$); **break**;
	...
	case first($term_n$): Pr($term_n$); **break**;
	default: error()
	}

Translation from regular languages to program code (2)

Assumptions :

- one symbol *lookahead*, stored in `sym`.

- `next ()` reads next symbol into `sym`.
- `error ()` quits with an error message.
- `first (exp)` is the set of *start symbols* of exp.

- The given syntax is assumed to be *left-parsable* (or : deterministic).

Translation from regular languages to program code (3)

This means :

K	Condition
$term_1$ | ... | $term_n$	The terms do not have any common start symbols.
$fact_1$ ... $fact_n$	if $fact_i$ contains the empty sequence then $fact_i$ and $fact_{i+1}$ do not have any common start symbols.
{exp}, [exp]	if exp contains the empty sequence then the set of start symbols of exp may not contain any symbol that can also follow it.

Example : A Scanner for Identifiers

```
void ident () {
  if (isLetter(ch)) next(); else error();
  while (isLetterOrDigit(ch)) {
    switch (ch) {
    case 'a': ... case 'z':
    case 'A': ... case 'Z': letter(); break;
    case '0': ... case '9': digit(); break;
    }
  }}

where

  boolean isLetter(char ch) {
    return
      'a' <= ch && ch <= 'z' || 'A' <= ch && ch <= 'Z '
  }

  boolean isDigit(char ch) {
    return '0' <= ch && ch <= '9';
  }

  boolean isLetterOrDigit(char ch) {
    return isLetter(ch) || isDigit(ch);
  }
```

```
void letter() {
    switch (ch) {
    case 'a': if (ch == 'a') next(); else error();
    ...
    case 'Z': if (ch == 'Z') next(); else error();
  }
void digit() {
    switch (ch) {
    case '0': if (ch == '0') next(); else error();
    ...
    case '9': if (ch == '9') next(); else error();
  }
```

- or, a little more streamlined:

```
void ident () {
  if ('a' <= ch && ch <= 'z' ||
      'A' <= ch && ch <= 'Z')
    next();
  else error();
  while ('a' <= ch && ch <= 'z' ||
         'A' <= ch && ch <= 'Z' ||
         '0' <= ch && ch <= '9')
   next();
```

The Task of a Lexical Analyzer

- The basic action of a lexical analyzer is to read some part of the input and to return a token:

```
Token sym;
void nextSym () {
   "skip white space and assign next token to sym"
}
```

- Whitespace can be
  - blank character, tabulator, newline
  - more general: any character <= ' '
  - comments: any sequence of characters enclosed in /* ... */.
- A token consists of a token class and possibly some additional information.

Whitespace and Tokens

- Token classes

IDENT	`foo, main,`
NUMBER	`0, 123, 1000`
FLOAT	`0.5   1.0e+3`
STRING	`"", "a", "*** error"`
MODULE	`module`
VOID	`void`
LPAREN	`(`
RPAREN	`)`
LBRACE	`{`
RBRACE	`}`
SEMICOLON	`;`
EOF	`\uFFFF     (i.e. (char)-1)`
...	

- Token classes are represented as int's in Java.

Example Run of a Lexical Analyzer

- For the following J0 program

```
module M {
  void main () {
    println ("hello world\n");
  }
}
```

- The lexical analyzer should return:

  MODULE IDENT(M) VOID IDENT(main) LPAREN
  RPAREN LBRACE IDENT(println) LPAREN
  STRING("hello world\n") RPAREN SEMICOLON
  RBRACE RBRACE EOF

The Interface of a Lexical Analyzer

```
class Scanner {
      /** Constructor */
      Scanner (InputStream in)

      /** The symbol read last */
      int sym;

      /** The symbol's character representation */
      String chars;

      /** Read next token into sym and chars */
      void nextSym ()

      /** Close input stream */
      void close()

 }
```

Lexical syntax of EBNF

The syntax of EBNF lexemes :

```
symbol        =  {blank}
                 (identifier | literal |
                  "(" | ")" | "[" | "]" | "{" | "}" | "|" | "=" | ". " ).
Identifier    =  letter { letter | digit }.
literal       =  "\"" {stringchar} "\" ».
stringchar    =  escapechar  |  plainchar.
escapechar    =  "\\" char.
plainchar     =  charNoQuote.
```

EBNF symbol definition

```
package ebnf;
    interface Symbols {
      static final int
        ERROR    = 0,
        EOF      = ERROR  +1, IDENT    = EOF    +1,
        LITERAL = IDENT  +1, LPAREN   = LITERAL+1,
        RPAREN   = LPAREN +1, LBRACK   = RPAREN +1,
        RBRACK   = LBRACK +1, LBRACE   = RBRACK +1,
        RBRACE   = LBRACE +1, BAR      = RBRACE +1,
        EQL      = BAR    +1, PERIOD   = EQL    +1;
    }
```

Java notes:

- Symbols kept in an interface which can be « inherited » by classes needing access to them.
- +1 trick compensates for lack of enums in Java.

EBNF Scanner (1)

```
package ebnf;
  import java.io.*;

class Scanner implements /*imports*/
  Symbols {

  /** the symbol recognized last */
  public int sym;

  /** if that symbol was an identifier
      or a literal, it's string
      representation */
  public String chars;

  /** the character stream being tokenized
  */
  private InputStream in;

  /** the next unconsumed character */
  private char ch;

  /** a buffer for assembling strings */
  private StringBuffer buf =
          new StringBuffer();

  /** the end of file character */
  private final char eofCh = (char) -1

  /** constructor */
  public Scanner(InputStream in) {
    this.in = in;
    nextCh(); }
```

```
  public static void error(String msg) {
    System.out.println(
          "**** error: "+msg );
    System.exit(-1);
  }

  /** print current character and read
      next character */
  private void nextCh() {
    System.out.print(ch);
    try {
      ch = (char)in.read();
    } catch (IOException ex) {
      error("read failure: " +
             ex.toString());
    }
  }

  /** read next symbol*/
  public void nextSym() {
    while (ch <= ' ') nextCh();
    switch (ch) {
      case 'a': . . . case'z':
      case 'A': . . . case'Z':
        buf.setLength(0);
        buf.append(ch); nextCh();
        while ('a' <= ch && ch <= 'z' ||
               'A' <= ch && ch <= 'Z' ||
               '0' <= ch && ch <= '9') {
          buf.append(ch); nextCh();}
```

EBNF Scanner (2)

```
      sym = IDENT;
      chars = buf.toString();
      break;
    case '\"':
      nextCh();
      buf.setLength(0);
      while (' ' <= ch && ch != eofCh &&
             ch != '\"') {
        if (ch == '\\') nextCh();
        buf.append(ch); nextCh();
      }
      if (ch == '\"') nextCh();
      else
        error("unclosed string literal");
      sym = LITERAL;
      chars = buf.toString();
      break;
    case '(':
      sym = LPAREN; nextCh(); break;
    case ')':
      sym = RPAREN; nextCh(); break;
    case '[':
      sym = LBRACK; nextCh(); break;
    case ']':
      sym = LBRACK; nextCh(); break;
    case '{':
      sym = LBRACE; nextCh(); break;
    case '}':
      sym = LBRACE; nextCh(); break;
```

```
    case '|':
      sym = BAR; nextCh(); break;
    case '=':
      sym = EQL; nextCh(); break;
    case '.':
      sym = PERIOD; nextCh(); break;
    case eofCh:
      sym = EOF; break;
    default:
      error("illegal character: " + ch +
      "(" + (int)ch + ")");
    }
  }

/** the string representation of a symbol*/
public static String representation
        (int sym) {
  switch (sym) {
  case ERROR   : return "<error>";
  case EOF     : return "<eof>";
  case IDENT   : return "identifier";
  case LITERAL: return "literal";
  case LPAREN  : return "`('";
  case RPAREN  : return "`)'";
    . . .
  default      : return "<unknown>"; }
}

public void close() throws IOException {
  in.close(); }
```

A Testbed for the EBNF Scanner

```
package ebnf;
import java.io.*;
class ScannerTest implements Symbols {
  static public void main(String[] args) {
    try {
      Scanner s = new Scanner(new FileInputStream(args[0]));
      s.nextSym();
      while (s.sym != EOF) {
        System.out.println("[" + Scanner.representation(s.sym) + "]");
        s.nextSym();
      }
      s.close();
    } catch (IOException ex) {
      System.out.println(ex);
      System.exit(-1);
    }
  }
}
```

The Longest Match Rule

- **Problem :**

  The given syntax for EBNF is ambiguous

  (why ?)

- **Solution :**

  The scanner matches at each step the *longest* symbol that fits the definition

  (« longest match rule »)

Generating Lexical Analyzers Automatically

- There is a systematic way to map any regular expression to a lexical analyzer
- Three steps:
  - Regular expression -> (nondeterministic) finite state automaton (NFA)
  - NFA -> deterministic finite state automaton (DFA)
  - DFA -> generated scanner program
- \- This can be automatized in a *scanner generator*.

Finite State Automata

- Consist of a finite number of *states* and *transitions*
- Transitions are labelled with input symbols
- There is one *start state*.
- A subset of states are the *final states*.
- A finite state automaton starts in the start state, and for each sinput symbol follows an edge labelled with that symbol.
- It *accepts* an input string iff it ends up in a final state.
- Examples: See blackboard, and Appel Figure 2.3.

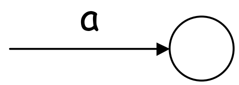
(Non)Deterministic Finite State Automata

- 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 follwed 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.

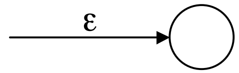
From Regular Expressions to NFA 's

- Here is a systematic way to translate any regular expression into an NFA :

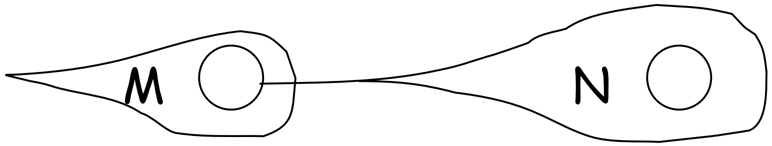
a



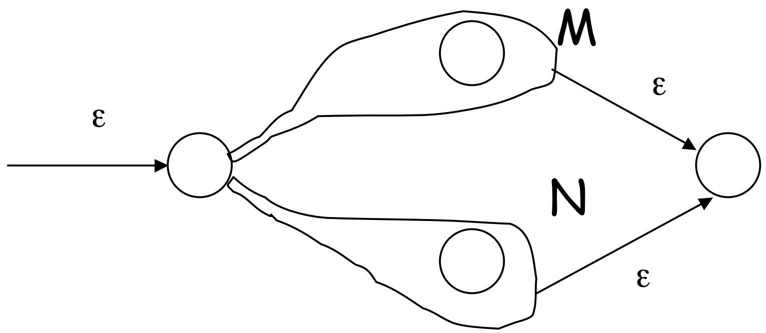
ε



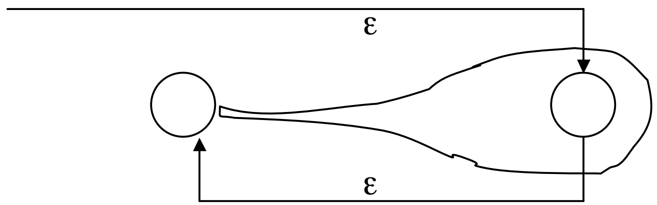
M N



M | N



{M}



Converting NFA 's to DFA 's

- Problem: Executing an NFA needs *backtracking*, which is inefficient.
- Would like to convert to a DFA
- Essential idea: Construct a DFA which has a state for each possible *set of states* a given NFA could be in.
- A set of states is final in a DFA if it contains a final NFA 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.

Algorithm to Convert NFA 's to DFA 's

- See Appel, Section 2.4
- First step: For a set of states S, let closure(S) be the smallest set of states that is reachable from S using only ε transitions.
- \- Algorithm to compute closure(S):

```
T := S
repeat
  T' := T ;
  for each state s in T
    for each edge e from s to some state s'
      if (e is labelled with ε)
        T := T ∪ {s '}
until T = T'
```

- 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 DFAedge

```
T := {}
for each state s in S
  for each edge e from s to some state s ’
    if (e is labelled with c)
      T := T \ ∪ closure({s'})
```

DFA Simulation

- Using the machinery developped so far, we can already *simulate* a DFA, given an equivalent NFA:
- Let s_1 be the NFA's start state and let the current input stream be c_1 ... c_k. Then the simulation works as follows:

```
d := closure({s_1})
for i := 1 to k do
  d := DFAedge (d, c_i)
```

- Manipulating these sets at run time is still very inefficient.

DFA Construction

- DFA states are numbered from 0
- 0 is the error state; the DFA goes into state 0 iff the NFA would have blocked because no edge matched the input symbol.
- Data structures:

  `states`: An array which maps each DFA state to the set of NFA states it represents.

  `trans`: A matrix of transitions from state numbers to state numbers

DFA Construction (2)

- Algorithm

```
states[0] := {}  // error state
states[1] := closure({s_1})
        j := 0 ; p := 2
/* states[0..j) have been processed completely
   states[j..p) are as yet unprocessed
*/
while j < p do
  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
```

Executing a DFA

- First possibility: Represent the DFA by a matrix:

  trans: Array [StateIndex, InputSymbol] of StateIndex

- Analyzer loop:

```
s := 0; // the DFA start state
while ("more input") {
  c := "next input character »
  s := trans[s, c]
}
```

Executing a DFA (2)

- Second possibility: Represent DFA by a case statement:

```
s := 0
while ("more input") {
  c := "next input character »
  switch (s) {
  case 0:
    switch (c) {
    case 'a': s := 3
    ...
    }
  ...
  }
}
```

Summary : Lexical Analysis

- Lexical analysis turns input charcaters 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.
  - This works if the grammar is left-parsable.
- By machine, going from regular expression to NFA to DFA.

Scanner generators

- There are a number of generators which generate a lexical analyzer automatically from a description.
- Description enumerates token classes and gives their syntax as regular expressions.
- Examples: Lex, JavaLex.
- Advantages of using a scanner generator?
- Disadvantages?