Lexical analysis

- The translation of a source program into tokens sequences is the main task of the *lexical analyzer* (*scanner*) of a compiler.
- The lexical syntax is usually described by a regular language.
- Hence lexical analyzers can be implemented by finite state machines.
- A scanner generator takes a grammar describing the lexical syntax and produces a lexical analyzer.

How does a SG work?

There is a systematic way to map any regular expression to a lexical analyzer:

1. Regular expression ⇒ nondeterministic finite state automaton (NFA).
2. NFA ⇒ deterministic finite state automaton (DFA).
3. DFA ⇒ scanner program.

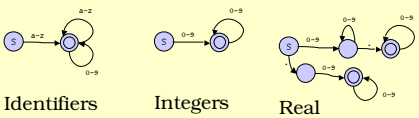
Finite State Automata

- A DFA consists of a finite number of *states* ( )and a set of *transitions*(→).
- Transitions are *labeled* with input characters (→).
- There is one *start state* (or).
- A subset of the states are called *final states* (or *accepting states*) ( ).

A DFA describes a language

- A DFA starts in the start state and for each input symbol follows an edge labeled with that symbol.
- It accepts an input string iff it ends in a final state.
- The language is made up of all accepted input strings.

Examples



Identifiers Integers Real numbers

Nondeterministic Finite State Automata

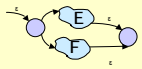
- In a NFA there can be more than one edge originating from the same node and labeled with the same label.
- Or there can be a special ε-edge, which can be followed without consuming any input.
- In contrast in a DFA all edges leaving a state have pair wise disjoint label sets and a input symbol is consumed on all transitions.

A reg-exp can be translated to a NFA

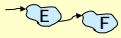
a

ε

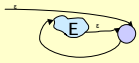
E|F



EF



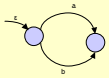
E*



E+ as EE*

E? as E|ε

[ab]



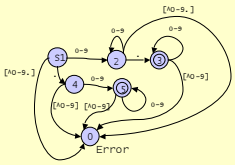
NFA to DFA

- Executing an NFA requires backtracking. 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 NFA states contains a final state.
  - Since the number of NFA states is finite, the number of DFA states is also finite (bounded by 2^N).
- See the book for details.

Coding a DFA

- Number each states from 0 (the error state).
- Create a multi dimensional array `trans` with the transitions from each state and for each input.
  `NextState = trans[State,ch]`
- If there is no transition for a given symbol in a given state: return 0.

Coding a DFA



Add an error state.

Number each state.

Create an array.

S\ch	0	1	2	3	4	5	6	7	8	9	.	*
0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	2	2	2	2	2	2	2	2	2	4	0
2	2	2	2	2	2	2	2	2	2	2	3	0
3	3	3	3	3	3	3	3	3	3	3	0	0
4	5	5	5	5	5	5	5	5	5	5	0	0
5	5	5	5	5	5	5	5	5	5	5	0	0

Executing a DFA

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


Executing a DFA

```
State = 1;
ch = nextCh();
while (ch != EOF) {
  switch (State) {
    case 0: error(); break;
    case 1: switch(ch) {
      case '0' : State = 2; break;
            …
      }
      …
  }
  ch = nextCh();
}
```

JLex

- As mentioned before we will use the scanner generator JLex in the project.
- JLex can be downloaded from:
  `http://www.cs.princeton.edu/~appel/modern/java/JLex`
- It will also be installed on the computers in the lab.

The input to JLex

- The input file (usually Name.lex) to JLex has three parts separated by a line with %%.

```
User code
%%
JLex directives
%%
Regular expression rules
```


A JLex example…

```
package expression;
import java_cup.runtime.*;
%%
%cup
%class Scanner
%eofval{
  return mkToken(EOF);
%eofval}
%line
%char
```

…

…a JLex example…

```
DIGIT =   [0-9]
WHITE =   " \t\n"
%%
"+"       {return mkToken(PLUS);}
…
{WHITE}   {/*Ignore white space*/}
{DIGIT}+ {return mkToken(INT,
           new integer(yytext()));}
.         {Report.error( … );}
```


… a JLex example

expr.lex:

```
package expression;
import java_cup.runtime.*;
%%
%cup
%class Scanner
%eofval{
  return mkToken(EOF);
%eofval}
%line
%char

DIGIT =   [0-9]
WHITE =   " \t\n"
%%
"+"       {return mkToken(PLUS);}
"-"       {return mkToken(MINUS);}

{WHITE}   {/*Ignore white space*/}
{DIGIT}+ {return mkToken(INT, new integer(yytext()));}
.         {System.err.println("Illegal character:  " + yytext()); }
```


Tokens

- Apart from the input file we need some code to interface with the rest of the compiler. First of all we need a definition of the tokens used:

Tokens.Java:

```
package expression;
interface Tokens {
  int EOF   = 0;
  int PLUS  = 1;
  int MINUS = 2;
  int INT   = 3;
}
```

java_cup.runtime.Symbol

- To use the scanner with the parser we will use later, we have to put the result in a format that the parser understands.
- The parser generator JavaCup provides the class Symbol, with the constructor:

```
Symbol(int sym, int left, int right, Object value)
```


Using Symbol

```
%{
private Symbol mkToken(int k) {
  return mkToken(k, null);
}

private Symbol mkToken(int k, Object value) {
  return new Symbol(k, yychar, yychar+yylength(),
                    value);
}
%}
```


The complete expr.lex

expr.lex:

```
package expression;
import java_cup.runtime.*;
%%
%cup
%class Scanner
%eofval{
  return mkToken(EOF);
%eofval}
%line
%char
%{
private Symbol mkToken(int k) { return mkToken(k, null);}
private Symbol mkToken(int k, Object value) {
  return new Symbol(k, yychar, yychar+yylength(), value);}
%}
DIGIT =  [0-9]
WHITE =  " \t\n"
%%
"+"      {return mkToken(PLUS);}
"-"      {return mkToken(MINUS);}
{WHITE}  {/*Ignore white space*/}
{DIGIT}+ {return mkToken(INT, new integer(yytext()));}
.        {System.err.println("Illegal character: " + yytext()); }
```


Summary: Lexical Analysis

- Lexical analysis turns input characters into tokens.
- Lexical analysis is described by regular expressions.
- Scanners can be created by hand or automatically from a reg-exp.