PART II
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.

Writing a Scanner

- Imagine we would like to write a scanner for binary numbers:

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

- We are given:
  - A variable (`ch`) containing a lookahed.
  - A function (`int nextCh()`) which read a character from the input.
  - And a function (`void error()`) which quits with an error message.

Coding a reg-exp

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|F	`if(ch in first(E)){Prog(E)} else {Prog(F)}`

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

What is First(BinNumber)?

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

- First(BinNumber) = First(BinDigit).
- First(BinDigit) = First("0") ∪ First("1").
- First(BinDigit) = {"0","1"}.
- First(BinNumber) = {"0","1"}.

```
Ch in first(BinDigit):
  (ch == '0' || ch == '1')
```


Straightforward Generation

```
BinNumber ::=
  BinDigit

  {BinDigit}



BinDigit ::=
  "0" | "1"
```

```
void BinNumber() {
  BinDigit
  {BinDigit}
}

void BinDigit() {
  "0" | "1"
}
```


Straightforward Generation

```
BinNumber ::=
  BinDigit

  {BinDigit}



BinDigit ::=
  "0" | "1"
```

```
void BinNumber() {
  BinDigit
  {BinDigit}
}

void BinDigit() {
  "0" | "1"
}
```

Straightforward Generation


```
BinNumber ::=
  BinDigit

  {BinDigit}



BinDigit ::=
   "0" | "1"
```

```
void BinNumber() {
  BinDigit();
  {BinDigit}
}

void BinDigit() {
  "0" | "1"
}
```

Straightforward Generation


```
BinNumber ::=
  BinDigit

  {BinDigit}



BinDigit ::=
   "0" | "1"
```

```
void BinNumber() {
  BinDigit();
  {BinDigit}
}

void BinDigit() {
  "0" | "1"
}
```

Straightforward Generation


```
BinNumber ::=
  BinDigit

  {BinDigit}



BinDigit ::=
   "0" | "1"
```

```
void BinNumber() {
  BinDigit();
   while (ch in first(BinDigit)) {
    BinDigit
  }
}

void BinDigit() {
  "0" | "1"
}
```

Straightforward Generation


```
BinNumber ::=
  BinDigit

  {BinDigit}



BinDigit ::=
   "0" | "1"
```

```
void BinNumber() {
  BinDigit();
   while (ch in first(BinDigit)) {
    BinDigit
  }
}

void BinDigit() {
  "0" | "1"
}
```

Straightforward Generation


```
BinNumber ::=
  BinDigit

  {BinDigit}



BinDigit ::=
   "0" | "1"
```

```
void BinNumber() {
  BinDigit();
   while (ch == '0' || ch == '1') {
    BinDigit();
  }
}

void BinDigit() {
  "0" | "1"
}
```

Straightforward Generation


```
BinNumber ::=
  BinDigit

  {BinDigit}



BinDigit ::=
   "0" | "1"
```

```
void BinNumber() {
  BinDigit();
   while (ch == '0' || ch == '1') {
    BinDigit();
  }
}

void BinDigit() {
  if (ch == '0') {
    "0"
  } else {
    "1"
  }
}
```

Straightforward Generation

```
BinNumber ::=
  BinDigit

  {BinDigit}



BinDigit ::=
   "0" | "1"
```

```
void BinNumber() {
   BinDigit();
   while (ch == '0' || ch == '1') {
     BinDigit();
   }
 }

void BinDigit() {
  if (ch == '0') {
     "0"
   } else {
     "1"
   }
 }
```


Straightforward Generation

```
BinNumber ::=
  BinDigit

  {BinDigit}



BinDigit ::=
   "0" | "1"
```

```
void BinNumber() {
   BinDigit();
   while (ch == '0' || ch == '1') {
     BinDigit();
   }
 }

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


Optimized version

```
BinNumber ::=
  BinDigit



  {BinDigit}



BinDigit ::=
   "0" | "1"
```

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

To optimize:

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

Possible problems

The method does not always work:

```
OneOrTwo ::= [BinDigit] BinDigit
void OneOrTwo() {
 if(ch =='0' || ch =='1') {ch=nextCh();}
 if(ch =='0' || ch =='1') {ch=nextCh();}
 else { error();}
```

```
OneOrMore ::= {BinDigit} BinDigit
IntOrFloat ::= Number | Number "." Number
```


Solution

- For regular expression these problems can be solved by rewriting the grammar.
- In general this is not the case if the grammar uses recursion.

```
OneOrTwo ::= BinDigit [BinDigit]

OneOrMore ::= BinDigit {BinDigit}

IntOrFloat ::= Number ["." Number]
```


The task of the Lexical Analyzer

- So far we have recognized only one kind of tokens at a time.
- Usually a scanner has to recognize a variety of tokens and return the one it found.
- A scanner also has to skip white space and comments.
- Fore some tokens the scanner also needs to collect additional information:
  - Which number was it?
  - The source position of the token.

An Example

```
3 * (5 + 3) /* comment */ - 7
```

The scanner should return:

```
INT(3) TIMES LPAR INT(5) PLUS
INT(3) RPAR MINUS INT(7)
```


The longest match rule

- When does one token end and the next token start?
- What does the following Java expressions mean?

Solution: The scanner matches the longest possible token.

`x +++ y`	x ++ + y, add, then increment x
`x + ++ y`	x + ++ y, increment y then add
`x ++++ y`	x ++ ++ y, invalid

Scanner Generators

- If we can describe all our tokens with regular expressions (which we usually can), then we can automatically generate a program that can do the recognition.
- We use a program, a scanner generator (SG), that takes a description of the tokens and generates a scanner that recognizes all tokens.
- This saves a lot tedious work in writing a scanner by hand.

Scanner Generators

- The input to a scanner generator consist of a list of pairs (*pattern*, *action*), e.g.:

```
[0-9]     { return mkToken(INT, input)}
"+"       { return mkToken(PLUS)}
```

- The SG uses this input to build a source file for the actual scanner.
- In our project we will use the SG JLex:
  `http://www.cs.princeton.edu/~appel/modern/java/JLex`

JLex

- The scanner will perform the action of the first matching pattern.
- The patterns are regular expressions.
- The actions are Java code, typically the action will return the corresponding token.
- The longest possible input string is matched.

Reg-exps in JLex

- Apart from the special caracters ? * + | ( ) ^ $ / ; . = < > [ ] { } " \ and blank, every character stands for itself.
- \ quotes a special character so that it loses its special meaning e.g. \+.
- Between two duble quotes "" all special characters, but \ and " looses their special meaning.
- The following escape sequences are recognized: \b \n \t \f \r.

Reg-exps in JLex

- | * ? + () [] have their *usual* meaning:

```
E* for {E}.       [ac] for ("a"|"c").
E? for [E].       [a-d] for ("a"|"b"|"c"|"d").
E+ for E{E}.      [^0-9] for anything but a digit.
```

- `Name = E` is used to define macros.
- {name} refers to the macro name.
- We can use . as a shortcut for [^\n]

Examples of reg-exps

`[0-9]+`

describes integer numbers (42, 17).

`\"[^\"\n]*\"`

describes simple strings without newlines ("foo", "@42 erik!").

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( … );}
```

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:

```
package expression;
interface Tokens {
  int EOF  = 0;
  int PLUS = 1;
  …
  int INT  = 7;
}
```

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)
```


The project and JLex

- More details about how to use JLex will be presented together with the project.
- A framework for using JLex will also be provided.
- For further information about JLex see the manual available on the course page.

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.

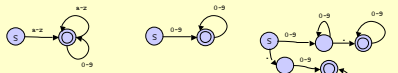
Finite State Automata

- 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

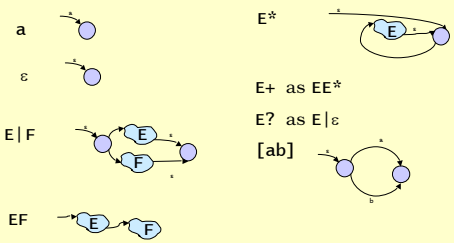


Identifiers | Integers | Real numbers

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



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.

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 'a' : State = 3; break;
            …
      }
      …
  }
  ch = nextCh();
}
```

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.