# Languages and Compilers (SProg og Oversættere)

# Lecture 6 Lexical Analysis

Bent Thomsen

Department of Computer Science

Aalborg University

# Learning goals

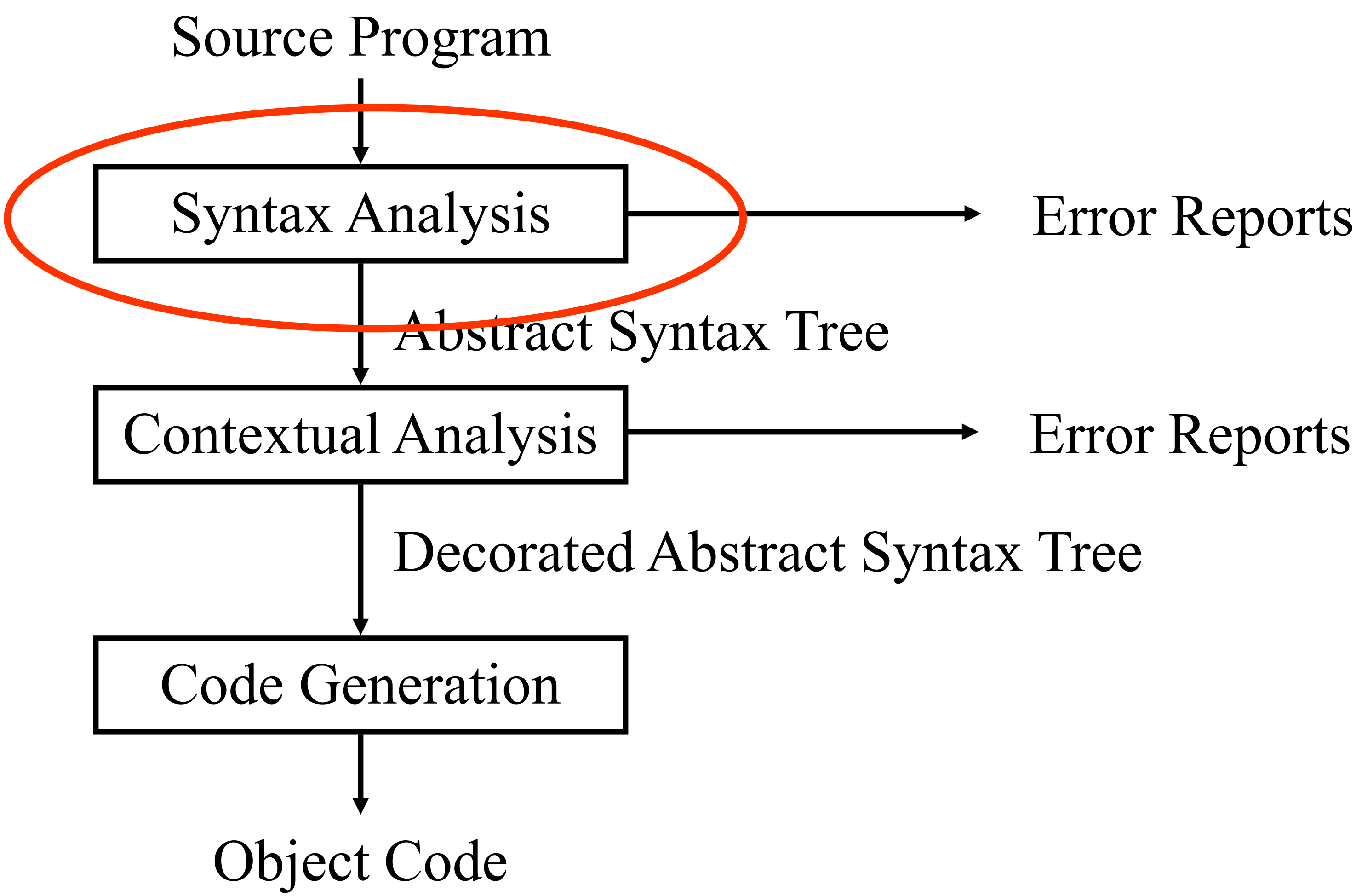
- Understand the lexical analysis phase of the compiler
- Understand the role of regular expressions
- Understand the structure of the lexical analysis
- Understand the role of finite automata
- Get an overview of the Jlex tool

# Remember exercise 4 from before lecture 1 ?

- Write a Java program that can read the string “a + n * 1” and produce a collection of objects containing the individual symbols when blank spaces are ignored (or used as separator).

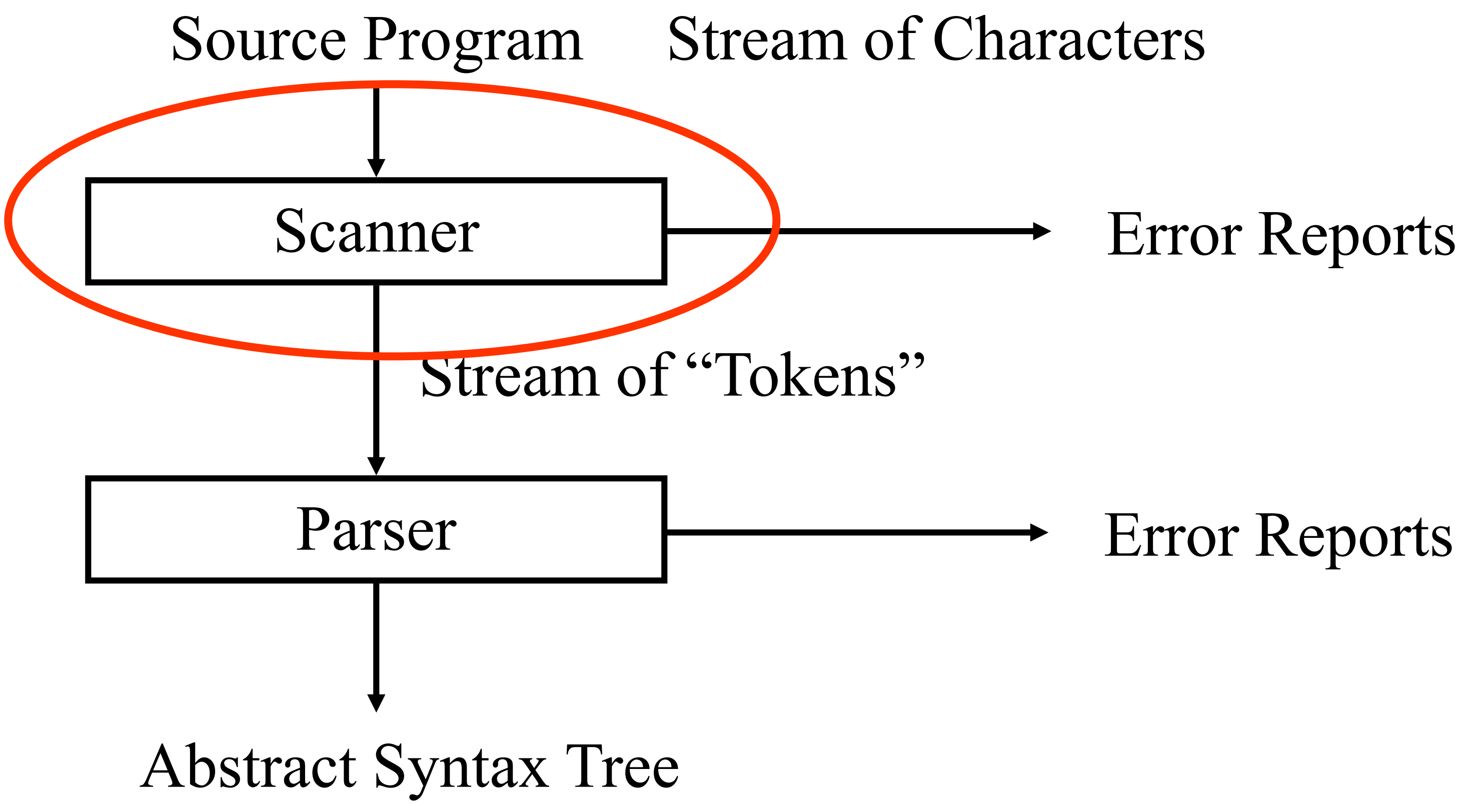
- Today we shall see several ways of solving this exercise

# The “Phases” of a Compiler



# Syntax Analysis: Scanner

**Dataflow chart**



# 1) Scan: Divide Input into Tokens

An example ac source program:

```
f b
i a
a = 5
b = a + 3.2
p b
```

**Lexems** are “words” in the input, for example  keywords, operators, identifiers, literals, etc.
**Tokens** is a datastructure for lexems and additional information

*scanner*

| *floatdl* | *id* | *intdcl* | *id* | *id* | *assign* | *inum* | |
|---|---|---|---|---|---|---|---|
| f | b | i | a | a | = | 5 | … |

| | *assign* | *id* | *plus* | *fnum* | *print* | *id* | *eot* |
|---|---|---|---|---|---|---|---|
| … | = | a | + | 3.2 | p | b | |

# Developing a Scanner

In Java the scanner will normally return instances of Token:

```
public class Token {
  byte kind; String spelling;
  final static byte
    IDENTIFIER = 0; INTLITERAL = 1; OPERATOR   = 2;
    BEGIN      = 3; CONST      = 4; ...
    ...

  public Token(byte kind, String spelling) {
    this.kind = kind; this.spelling = spelling;
}


  ...
}
```

# 1) Scan: Divide Input into Tokens

An example ac source program:

```
f b
i a
a = 5
b = a + 3.2
p b
```

**Lexems** are “words” in the input, for example  keywords, operators, identifiers, literals, etc.
**Tokens** is a datastructure for lexems and additional information

*scanner*

| *floatdl* | *id* | *intdcl* | *id* | *id* | *assign* | *inum* | |
|---|---|---|---|---|---|---|---|
| | `b` | | `a` | `a` | | `5` | … |

| | *assign* | *id* | *plus* | *fnum* | *print* | *id* | *eot* |
|---|---|---|---|---|---|---|---|
| … | | `a` | | `3.2` | | `b` | |

# Developing a Scanner

In Java the scanner will normally return instances of Token, but we could also use a subclass hierachy:

```
abstract class Token ..

public class IdentToken extends Token {
  String spelling;
...

  public IdentToken(String spelling) {
    this.spelling = spelling;
}

public class AssignToken extends Token {


  ...
}
```

# Programming Language Specification

- A Language specification has (at least) three parts
  - Syntax of the language:
    - **Lexems/tokens as regular expressions**
      - **Reserved words**
    - Grammar (CFG) - usually formal in BNF or EBNF
  - Contextual constraints:
    - scope rules (often written in English, but can be formal)
    - type rules (formal or informal)
  - Semantics:
    - defined by the implementation
    - informal descriptions in English
    - formal using operational or denotational semantics

# Lexical Elements

- Character set
  - Ascii vs Unicode
- Identifiers
  - Java vs C#
- Operators
  - +, -, /, * , …
- Keywords
  - If, then, while
- Noise words
- Elementary data
  - numbers
    - integers
    - floating point
  - strings
  - symbols
- Delimiters
  - Begin .. End vs {…}
- Comments
  - /*  vs. # vs. !
- Blank space
- Layout
  - Free- and fixed-field formats

# Java Keywords

abstract continue for new switch assert default if package synchronized boolean do goto private this break double implements protected throw byte else import public throws case enum instanceof return transient catch extends int short try char final interface static void class finally long strictfp volatile const float native super while

- The keywords const and goto are reserved, even though they are not currently used.
- While true and false might appear to be keywords, they are technically Boolean literals
- Similarly, while null might appear to be a keyword, it is technically the null literal

# Lexems

- The Lexem structure can be more detailed and subtle than one might expect
  - String constants: “”
    - Escape sequence: \”, \n, …
    - Null string
  - Rational constants
    - 0.1, 10.01,
    - .1, 10. vs. 1..10

- Design guideline:
  - if the lexem structure is complex then examine the language for design flaws !!

- Note recent research shows huge difference between novices and experienced programmers views on keywords:
  - repeat while … do .. end vs. while (..) {…}

# (Try to) Avoide Weird Stuff

- PL/I
  - IF IF = THEN THEN = ELSE; ELSE ELSE = END; END
- C#
  - if (@if == then) then = @else; else @else = end;
- C
  - a (* b) … call of a with pointer to b or declaration on pointer b to a type where a is defined using typedef
- Whitespace language
  - Commands composed of sequences of spaces, tab stops and linefeeds

# Simple grammar for Identifiers

**Example:**

```
Start ::= Letter
        | Start Letter
        | Start Digit
Letter ::= a | b | c | d | ... | z
Digit  ::= 0 | 1 | 2 | ... | 9
```

This grammar can be transformed to a regular expression:

[a-z]([a-z]|[0-9])*

# Regular Expressions

| | |
|---|---|
| ε | The empty string |
| $t$ | Generates only the string $t$ |
| $X\ Y$ | Generates any string $xy$ such that $x$ is generated by $x$ and $y$ is generated by $Y$ |
| $X \mid Y$ | Generates any string which generated either by $X$ or by $Y$ |
| $X^*$ | The concatenation of zero or more strings generated by $X$ |
| $(X)$ | For grouping |

# Identifier Grammar Easily Transform to RE

Elimination of Left Recursion

$N ::= X \mid N\ Y \Rightarrow N ::= X\ Y*$

Left factorization

$X\ Y \mid X\ Z \Rightarrow X(Y|Z)$

**Example:**

```
Identifier ::= Letter
             | Identifier Letter
             | Identifier Digit
```

```
Identifier ::= Letter
             | Identifier (Letter|Digit)
```

```
Identifier ::= Letter (Letter|Digit)*
```

# Regular Grammers

- A grammar is regular if by substituting every nonterminal (except the root one) with its righthand side, you can reduce it down to a single production for the root, with only terminals and operators on the right-hand side.
- I.e. this grammer is regular:

```
Identifier ::= Letter
             | Identifier Letter
             | Identifier Digit
```

- Because it can be reduced to:

```
Identifier ::= Letter (Letter|Digit)*
```

# Regular Grammers

- Or rather

```
( a | b | c | d | ... | z )((a | b | c | d | ... | z)|(0 | 1 | 2 | ... | 9 ))*
```

- Which is called a regular expression, often written as:

```
[a-z]([a-z]|[0-9])*
```

- Sometimes regular grammers are described as:
  - Right regular i.e. having the form A := a A | b
  - Left regular i.e. having the form A := A a | b

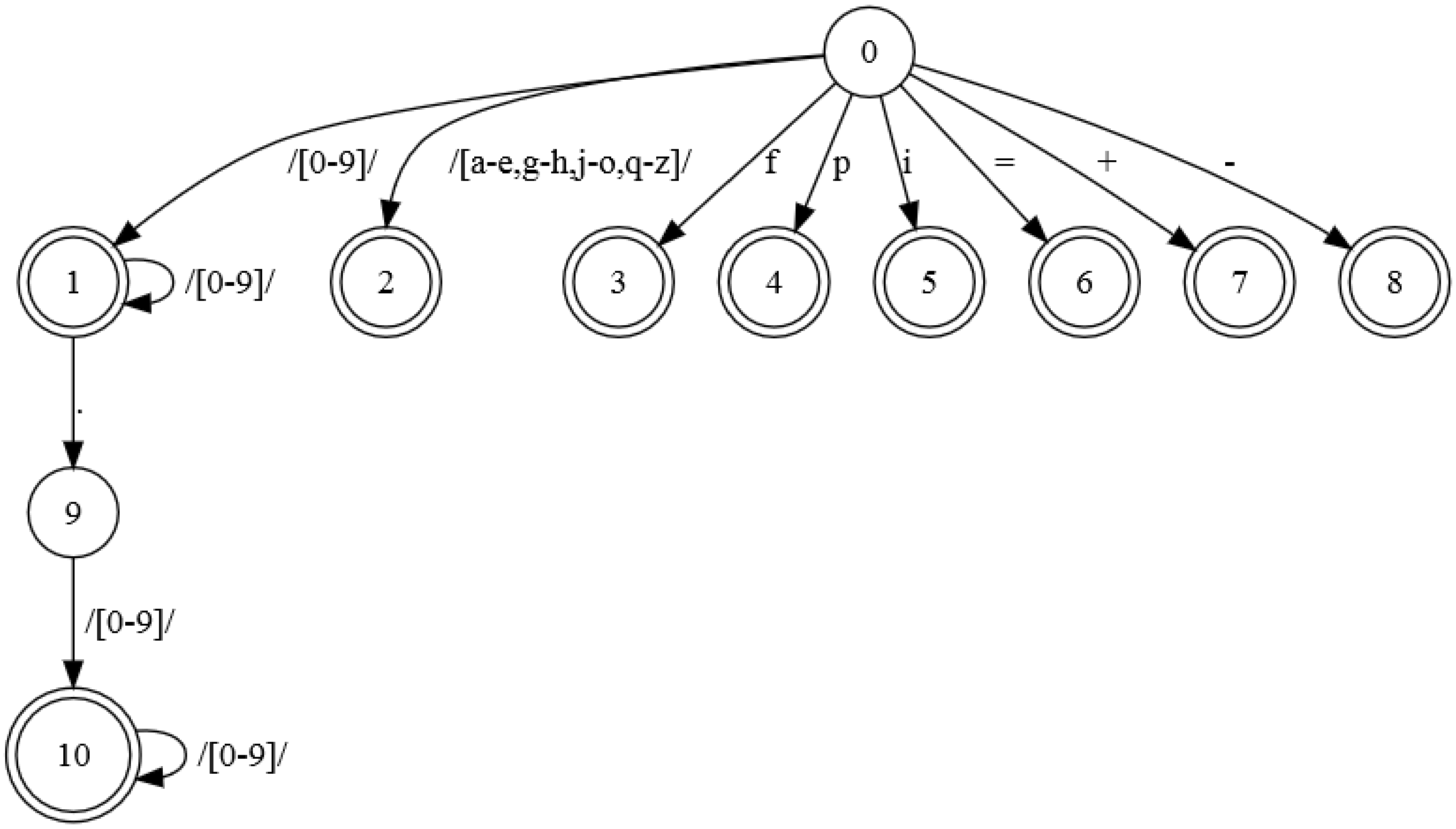
- Why are we so interested in Regular Expressions?
  - Because there are simple implementation techniques for Res
  - REs can be implemented via Finite State Machines (FSM)

# ac Token Specification

| Terminal | Regular Expression |
|---|---|
| floatdcl | "f" |
| intdcl | "i" |
| print | "p" |
| id | $[a - e] \mid [g - h] \mid [j - o] \mid [q - z]$ |
| assign | "=" |
| plus | "+" |
| minus | "-" |
| inum | $[0 - 9]^+$ |
| fnum | $[0 - 9]^+ . [0 - 9]^+$ |
| blank | $(" \; ")^+$ |

Figure 2.3: Formal definition of ac tokens.

[0-9]+ | [0-9]+.[0-9]+ | [a-e,g-h,j-o,q-z] | f | p | i | = | \+ | -



```
function SCANNER( ) returns Token
    while s.PEEK( ) = blank do call s.ADVANCE( )
    if s.EOF( )
    then ans.type ← $
    else
        if s.PEEK( ) ∈ { 0, 1, . . . , 9 }
        then ans ← SCANDIGITS( )
        else
            ch ← s.ADVANCE( )
            switch (ch)
                case { a, b, . . . , z } − { i, f, p }
                    ans.type ← id
                    ans.val ← ch
                case f
                    ans.type ← floatdcl
                case i
                    ans.type ← intdcl
                case p
                    ans.type ← print
                case =
                    ans.type ← assign
                case +
                    ans.type ← plus
                case -
                    ans.type ← minus
                case default
                    call LEXICALERROR( )
    return (ans)
end
```

Figure 2.5: Scanner for the ac language. The variable $s$ is an input stream of characters.

```java
/**
 * Figure 2.5 code, processes the input stream looking
 *   for the next Token.
 * @return the next input Token
 */
public static Token Scanner() {
    Token ans;
    while (s.peek() == BLANK)
        s.advance();
    if (s.EOF())
        ans = new Token(EOF);
    else {
        if (isDigit(s.peek()))
            ans = ScanDigits();
        else {
            char ch = s.advance();

            switch(representativeChar(ch)) {
            case 'a':  // matches {a, b, ..., z} - {f, i, p}
                ans = new Token(ID, ""+ch); break;
            case 'f':
                ans = new Token(FLTDCL);  break;
            case 'i':
                ans = new Token(INTDCL);    break;
            case 'p':
                ans = new Token(PRINT);     break;
            case '=':
                ans = new Token(ASSIGN);    break;
            case '+':
                ans = new Token(PLUS);      break;
            case '-':
                ans = new Token(MINUS);     break;
            default:
                throw new Error("Lexical error on character with decimal value: " + (int)ch);

            }
        }

    }
    return ans;
}

/**
```

```
function ScanDigits( ) returns token
    tok.val ← " "
    while s.peek( ) ∈ {0, 1, . . . , 9} do
        tok.val ← tok.val + s.advance( )
    if s.peek( ) ≠ "."
    then  tok.type ← inum
    else
        tok.type ← fnum
        tok.val ← tok.val + s.advance( )
        while s.peek( ) ∈ {0, 1, . . . , 9} do
            tok.val ← tok.val + s.advance( )
    return (tok)
end
```

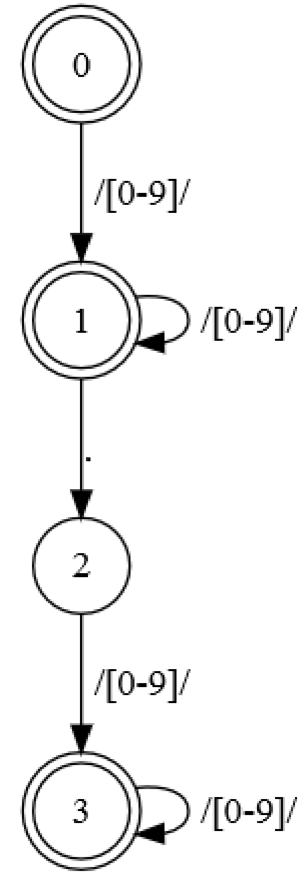



Figure 2.6: Finding inum or fnum tokens for the ac language.

```java
/**
 * Figure 2.6 code, processes the input stream to form
 *    a float or int constant.
 * @return the Token representing the discovered constant
 */

private static Token ScanDigits() {
    String val = "";
    int   type;
    while (isDigit(s.peek())) {
        val = val + s.advance();
    }
    if (s.peek() != '.')
        type = INUM;
    else {
        type = FNUM;
        val = val + s.advance();
        while (isDigit(s.peek())) {
            val = val + s.advance();
        }
    }
    return new Token(type, val);
}
```

# How to change code to accept: 0 | [1-9][0-9]*(.[0-9]*)

```
function ScanDigits( ) returns token
    tok.val ← " "
    while s.peek( ) ∈ { 0, 1, . . . , 9 } do
        tok.val ← tok.val + s.advance( )
    if s.peek( ) ≠ "."
    then  tok.type ← inum
    else
        tok.type ← fnum
        tok.val ← tok.val + s.advance( )
        while s.peek( ) ∈ { 0, 1, . . . , 9 } do
            tok.val ← tok.val + s.advance( )
    return (tok)
end
```

Figure 2.6: Finding inum or fnum tokens for the ac language.

# Pause

# Implement Scanner based on RE by hand

1) Express the “lexical” grammar as RE

   (sometimes it is easier to start with a BNF or an EBNF and do necessary transformations)

- For each variant make a switch on the first character by peeking the input stream
- For each repetition (..)* make a while loop with the condition to keep going as long as peeking the input still yields an expected character
- Sometimes the “lexical” grammar is not reduced to one single RE but a small set of REs – in this case a switch or if-then-else case analysis is used to determine which rule is being recognized, before following the first two steps

# Developing a Scanner

- Express the “lexical” grammar in EBNF

```
Token ::= Identifier | Integer-Literal | Operator |
        ; | : | := | ~ | ( | ) | eot
Identifier ::= Letter (Letter | Digit)*
Integer-Literal ::= Digit Digit*
Operator ::= + | - | * | / | < | > | =
Separator ::= Comment | space | eol
Comment ::= ! Graphic* eol
```

Now perform substitution and left factorization...

```
Token ::= Letter (Letter | Digit)*
      | Digit Digit*
      | + | - | * | / | < | > | =
      | ; | : (=|ε) | ~ | ( | ) | eot
Separator ::= ! Graphic* eol | space | eol
```

```
Token ::= Letter (Letter | Digit)*
        | Digit Digit*
        | + | - | * | / | < | > | =
        | ; | : (=|ε) | ~ | ( | ) | eot
```

```
private byte scanToken() {
  switch (currentChar) {
     case 'a': case 'b': ... case 'z':
     case 'A': case 'B': ... case 'Z':
        scan Letter (Letter | Digit)*
        return Token.IDENTIFIER;
     case '0': ... case '9':
        scan Digit Digit*
        return Token.INTLITERAL ;
     case '+': case '-': ... : case '=':
        takeIt();
        return Token.OPERATOR;
     ...etc...
}
```

# Developing a Scanner

Let's look at the identifier case in more detail

```
    ...
    return ...
  case 'a': case 'b': ... case 'z':
  case 'A': case 'B': ... case 'Z':
    acceptIt();
    while (isLetter(currentChar)
           || isDigit(currentChar) )
      acceptIt();
    return Token.IDENTIFIER;
  case '0': ... case '9':
    ...
```

Thus developing a scanner is a mechanical task.

# Developing a Scanner

In Java the scanner will normally return instances of Token:

```
public class Token {
  byte kind; String spelling;
  final static byte
    IDENTIFIER = 0; INTLITERAL = 1; OPERATOR   = 2;
    BEGIN      = 3; CONST      = 4; ...
    ...

  public Token(byte kind, String spelling) {
    this.kind = kind; this.spelling = spelling;
    if spelling matches a keyword change my kind
    automatically
  }

  ...
}
```

# Developing a Scanner

The scanner will return instances of Token:

```
public class Token {
...
  public Token(byte kind, String spelling) {
      if (kind == Token.IDENTIFIER) {
          int currentKind = firstReservedWord;
          boolean searching = true;
          while (searching) {
                  int comparison = tokenTable[currentKind].compareTo(spelling);
                  if (comparison == 0) {
                   this.kind = currentKind;
                  searching = false;
                  } else if (comparison > 0 || currentKind == lastReservedWord) {
                          this.kind = Token.IDENTIFIER;
                           searching = false;
                  } else {        currentKind ++;       }
          }
          } else
                  this.kind = kind;
 ...
```

# Developing a Scanner

The scanner will return instances of Token:

```
public class Token {
...

        private static String[] tokenTable = new String[] {
        "<int>",   "<char>",   "<identifier>",   "<operator>",
        "array",   "begin",   "const",   "do",   "else",   "end",
        "func",   "if",   "in",   "let",   "of",   "proc",   "record",
        "then",   "type",   "var",   "while",
        ".",   ":",   ";",   ",",   ":=",   "~",   "(",   ")",   "[",   "]",   "{",   "}",   ""
        "<error>"  };

        private final static int firstReservedWord = Token.ARRAY,
                                 lastReservedWord  = Token.WHILE;

...
}
```

# Developing a Scanner

Alternative implementation recognizing reserved words

```
  ...
  return ...
case 'i': acceptIt(); if (currentChar == 'f')  {acceptIt(); return Token.IF }
                      else if (currentChar == 'n')  {acceptIt(); return Token.IN }
…
case 'a': case 'b': ... case 'z':
case 'A': case 'B': ... case 'Z':
  acceptIt();
  while (isLetter(currentChar)
         || isDigit(currentChar) )
    acceptIt();
  return Token.IDENTIFIER;
case '0': ... case '9':
  ...
```

Thus developing a scanner is a mechanical task.

# Developing a Scanner

- Developing a scanner by hand is relatively easy for simple token grammars
- But for complex token grammars it can be hard and error prone
- The task can be automated
- Programming scanner generator is an example of declarative programming
  - What to scan, not how to scan
- Most compilers are developed using a generated scanner
- But before we look at doing that, we need some theory!

# FA and the implementation of Scanners

- Regular expressions, (N)DFA-ε and NDFA and DFA's are all equivalent formalism in terms of what languages can be defined with them.
- Regular expressions are a convenient notation for describing the "tokens" of programming languages.
- Regular expressions can be converted into FA's (the algorithm for conversion into NDFA-ε is straightforward)
- DFA's can be easily implemented as computer programs.

will explain this in subsequent slides

# Generating Scanners

- Generation of scanners is based on
  - Regular Expressions: to describe the tokens to be recognized
  - Finite State Machines: an execution model to which RE's are "compiled"

**Recap: Regular Expressions**

| | |
|---|---|
| ε | The empty string |
| *t* | Generates only the string *t* |
| *X Y* | Generates any string *xy* such that *x* is generated by *x* and *y* is generated by *Y* |
| *X* \| *Y* | Generates any string which generated either by *X* or by *Y* |
| *X** | The concatenation of zero or more strings generated by *X* |
| (*X*) | For grouping |

# Generating Scanners

- Regular Expressions can be recognized by a finite state machine. (often used synonyms: finite automaton (acronym FA))

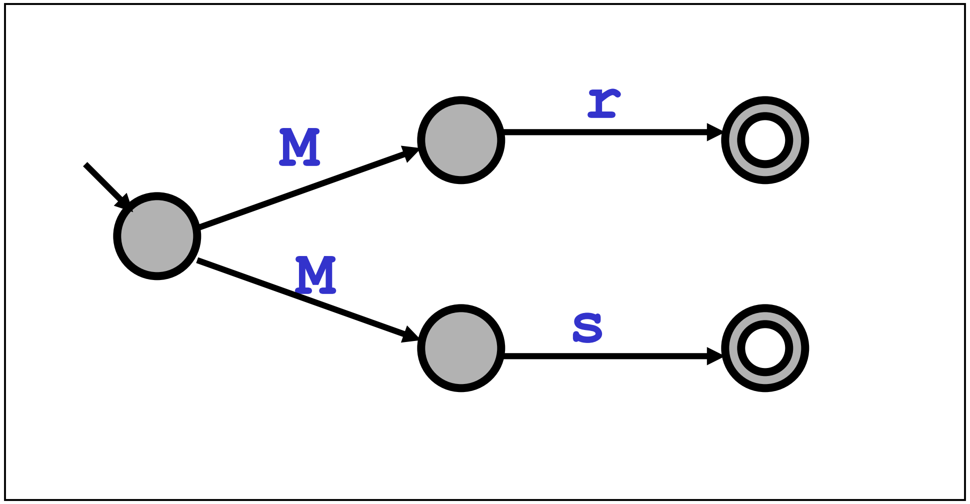
**Definition:** A finite state machine is an N-tuple (*States*,$\Sigma$,*start*,$\delta$,*End*)

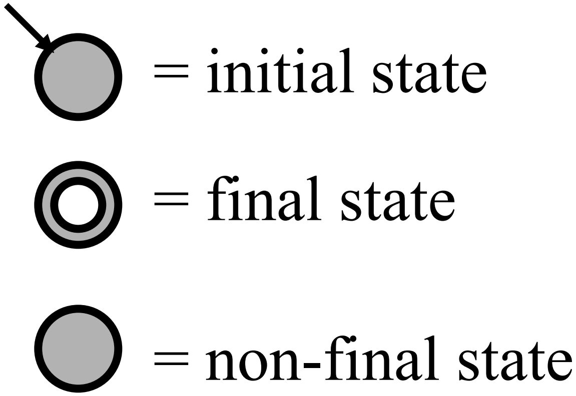
| | |
|---|---|
| *States* | A finite set of "states" |
| $\Sigma$ | An "alphabet": a finite set of symbols from which the strings we want to recognize are formed (for example: the ASCII char set) |
| *start* | A "start state" $Start \in States$ |
| $\delta$ | Transition relation $\delta \subseteq States \times States \times \Sigma$. These are "arrows" between states labeled by a letter from the alphabet. |
| *End* | A set of final states. $End \subseteq States$ |

# Generating Scanners

- Finite state machine: the easiest way to describe a Finite State Machine (FSM) is by means of a picture:

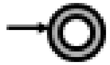
**Example:** an FA that recognizes `M r | M s`



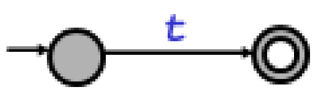



# Converting a RE into an NDFA-ε

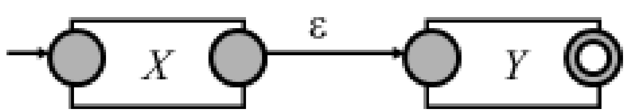
**RE:** ε
**FA:**



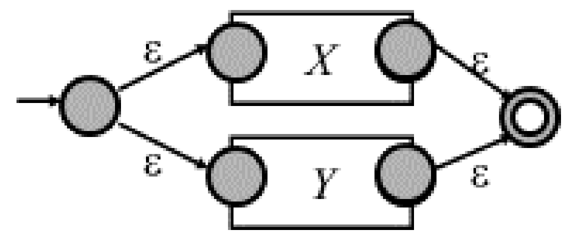
**RE:** t
**FA:**



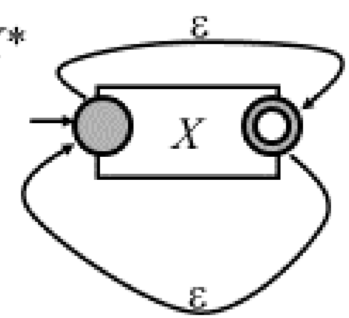

**RE:** *XY*
**FA:**




**RE:** *X*|*Y*
**FA:**



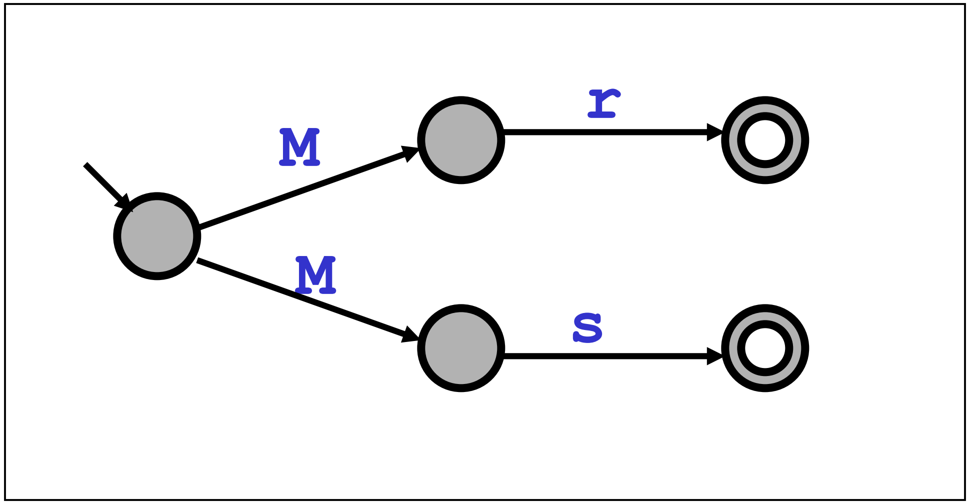

**RE:** *X**
**FA:**



# Deterministic, and non-deterministic FA

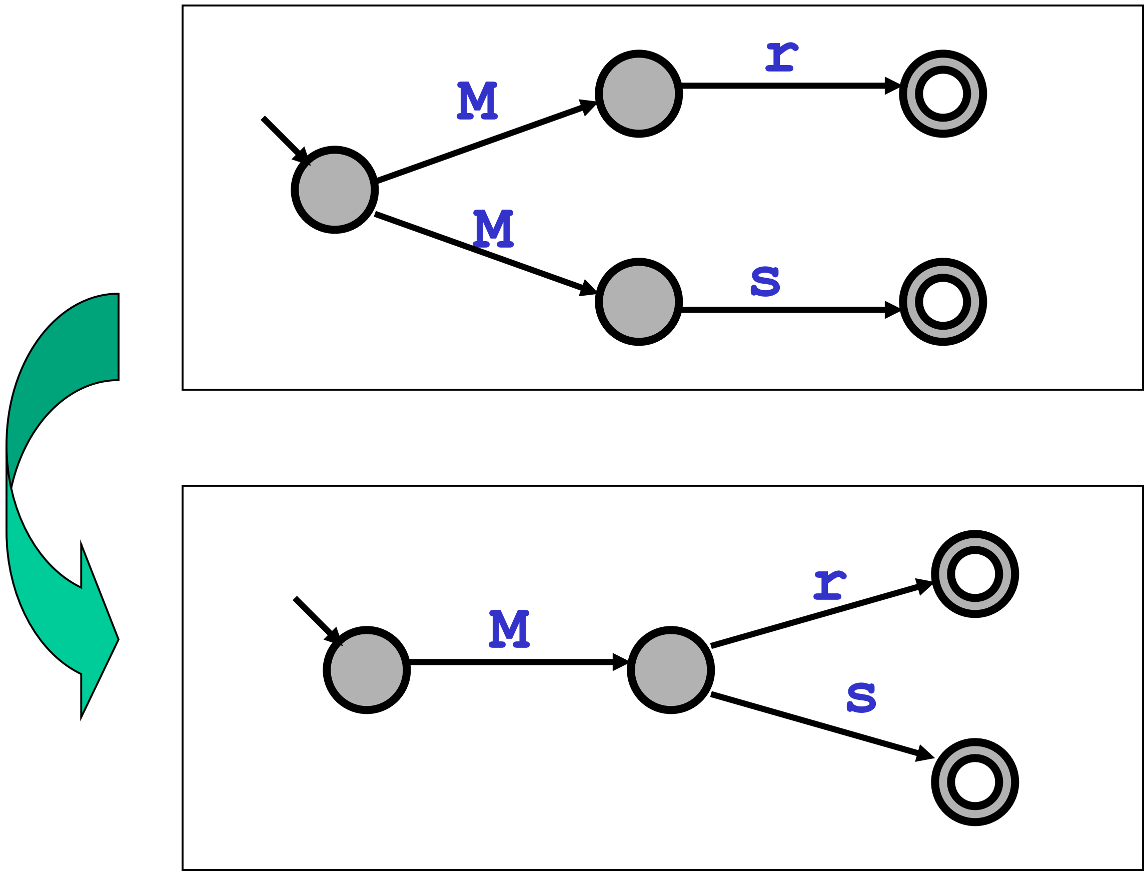
- An FA is called deterministic (acronym: DFA) if for every state and every possible input symbol, there is only one possible transition to choose from. Otherwise it is called non-deterministic (NDFA).

**Q:** Is this FSM deterministic or non-deterministic:



# Deterministic, and non-deterministic FA

- Theorem: every NDFA can be converted into an equivalent DFA.



```
function MakeDeterministic(N) returns DFA
    D.StartState ← RecordState({ N.StartState })
    foreach S ∈ WorkList do
        WorkList ← WorkList − { S }
        foreach c ∈ Σ do  D.T(S, c) ← RecordState( ⋃_{s∈S} N.T(s, c))
    D.AcceptStates ← { S ∈ D.States | S ∩ N.AcceptStates ≠ ∅ }
end

function Close(S, T) returns Set
    ans ← S
    repeat
        changed ← false
        foreach s ∈ ans do
            foreach t ∈ T(s, λ) do
                if t ∉ ans
                then
                    ans ← ans ∪ { t }
                    changed ← true
    until not changed
    return (ans)
end

function RecordState(s) returns Set
    s ← Close(s, N.T)
    if s ∉ D.States
    then
        D.States ← D.States ∪ { s }
        WorkList ← WorkList ∪ { s }
    return (s)
end
```

Figure 3.23: Construction of a DFA $D$ from an NFA $N$.

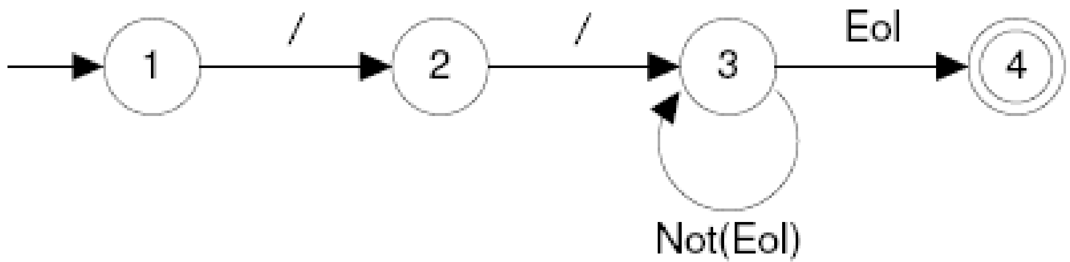
# Implementing a DFA

**Definition:** A finite state machine is an N-tuple (*States*, $\Sigma$, *start*, $\delta$, *End*)

| | |
|---|---|
| *States* | N different states => integers {0,..,N-1} => **`int`** data type |
| $\Sigma$ | **`byte`** or **`char`** data type. |
| *start* | An integer number |
| $\delta$ | Transition relation $\delta \subseteq States \times \Sigma \times States$. For a DFA this is a function $States \times \Sigma \rightarrow States$ Represented by a two dimensional array (one dimension for the current state, another for the current character. The contents of the array is the next state. |
| *End* | A set of final states. Represented (for example) by an array of booleans (mark final state by true and other states by false) |

Comment -> //(Not(Eol))*Eol

(a)




(b)

| State | Character | | | | |
|---|---|---|---|---|---|
| | / | Eol | a | b | ... |
| 1 | 2 | | | | |
| 2 | 3 | | | | |
| 3 | 3 | 4 | 3 | 3 | 3 |
| 4 | | | | | |

Figure 3.2: DFA for recognizing a single-line comment. (a) transition diagram; (b) corresponding transition table.

```
/*   Assume CurrentChar contains the first character to be scanned   */
State ← StartState
while true do
    NextState ← T[State, CurrentChar]
    if NextState = error
    then break
    State ← NextState
    CurrentChar ← READ( )
if State ∈ AcceptingStates
then   /* Return or process the valid token */
else   /* Signal a lexical error */
```

Figure 3.3: Scanner driver interpreting a transition table.

# Implementing a Scanner as a DFA

Slightly different from previously shown implementation (but similar in spirit):

- Not the goal to match entire input
  => when to stop matching?
  - Token(if), Token(Ident i) vs. Token(Ident ifi)

  Match longest possible token

  Report error (and continue) when reaching error state.
- How to identify matched token class (not just true|false)

  Final state determines matched token class

# FA and the implementation of Scanners

**What a typical scanner generator does:**

Token definitions
*Regular expressions*

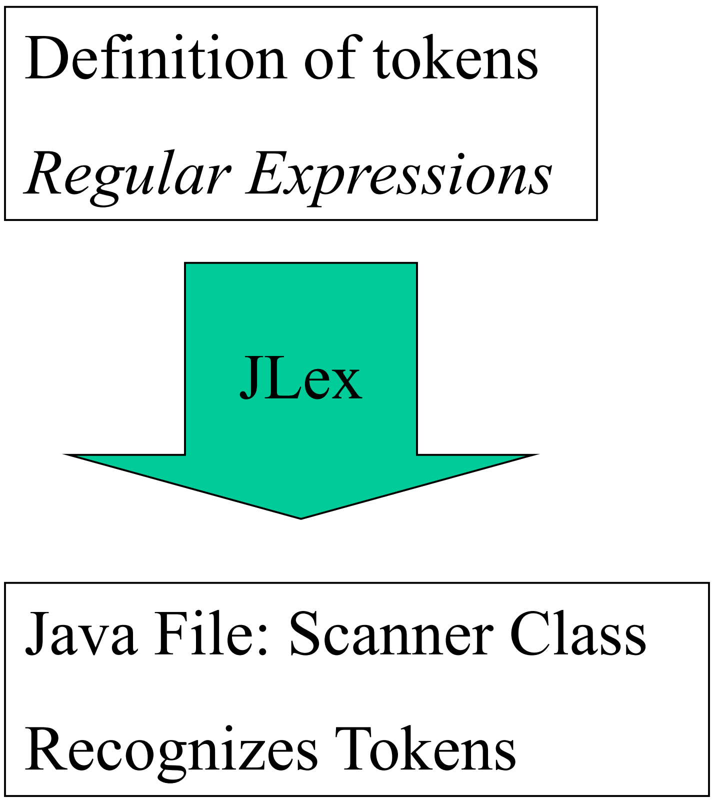
Scanner Generator

Scanner DFA
*Java or C or ...*

A possible algorithm:
- Convert RE into NDFA-ε
- Convert NDFA-ε into NDFA
- Convert NDFA into DFA
- generate Java/C/... code

**note:** In practice this exact algorithm is not used. For reasons of performance, sophisticated optimizations are used.
• direct conversion from RE to DFA
• minimizing the DFA

# JLex Lexical Analyzer Generator for Java




Writing scanners is a rather "robotic" activity which can be automated.

We will look at an example JLex specification (adopted from the manual).

Consult the manual for details on how to write your own JLex specifications.

# The JLex tool

Layout of JFLex file:

*user code (added to start of generated file)*

User code is copied directly into the output class

%%

*options*

JLex directives allow you to include code in the lexical analysis class, change names of various components, switch on character counting, line counting, manage EOF, etc.

%{

*user code (added inside the scanner class declaration)*

%}

*macro definitions*

Macro definitions gives names for useful regexps

%%

*lexical declaration*

Regular expression rules define the tokens to be recognised and actions to be taken

# JLex Regular Expressions

- Regular expressions are expressed using ASCII characters (0 – 127) or UNICODE using the `%unicode` directive.
- The following characters are *metacharacters*.

```
? * + | ( ) ^ $ . [ ] { } " \
```

- Metacharacters have special meaning; they do not represent themselves.
- All other characters represent themselves.

# JLex Regular Expressions

- Brackets `[ ]` match any single character listed within the brackets.
  - `[abc]` matches `a` or `b` or `c`.
  - `[A-Za-z]` matches any letter.
- If the first character after `[` is `^`, then the brackets match any character *except* those listed.
  - `[^A-Za-z]` matches any non-letter.
- Some escape sequences.
  - `\n` matches newline.
  - `\b` matches backspace.
  - `\r` matches carriage return.
  - `\t` matches tab.
  - `\f` matches formfeed.
- If `c` is not a special escape-sequence character, then `\c` matches `c`.

# JLex Regular Expressions

- Let `r` and `s` be regular expressions.
- `r?` matches *zero or one* occurrences of `r`.
- `r*` matches *zero or more* occurrences of `r`.
- `r+` matches *one or more* occurrences of `r`.
- `r|s` matches `r` or `s`.
- `rs` matches `r` concatenated with `s`.

- Parentheses are used for grouping.

```
("+"|"-")?
```

- Regular expression beginning with `^` is matched only at the beginning of a line.
- Regular expression ending with `$` is matched only at the end of a line.
- The dot `.` matches any non-newline character.

```
%%
[a-eghj-oq-z]        { return(ID); }
%%
```

Figure 3.10: A Lex definition for ac's identifiers.

```
%%
(" ")+                          { /* delete blanks */}
f                               { return(FLOATDCL); }
i                               { return(INTDCL); }
p                               { return(PRINT); }
[a-eghj-oq-z]                   { return(ID); }
([0-9]+)|([0-9]+"."[0-9]+)      { return(NUM);  }
"="                             { return(ASSIGN); }
"+"                             { return(PLUS); }
"-"                             { return(MINUS); }
%%
```

Figure 3.11: A Lex definition for ac's tokens.

```
%%
Blank                                 " "
Digits                                [0-9]+
Non_f_i_p                             [a-eghj-oq-z]
%%
{Blank}+                              { /* delete blanks */}
f                                     { return(FLOATDCL); }
i                                     { return(INTDCL); }
p                                     { return(PRINT); }
{Non_f_i_p}                           { return(ID); }
{Digits}|({Digits}"."{Digits})        { return(NUM);  }
"="                                   { return(ASSIGN); }
"+"                                   { return(PLUS); }
"-"                                   { return(MINUS); }
%%
```

Figure 3.12: An alternative definition for ac's tokens.

# Jlex for ac

```
package acFLEXCUP;

import java_cup.runtime.*;
import java.io.IOException;

import .AcLEXSym;
import static .AcLEXSym.*;

%%

%class AcLEXLex

%unicode
%line
%column

// %public
%final
// %abstract

%cupsym .AcLEXSym
%cup
// %cupdebug

%init{
    // TODO: code that goes to constructor
%init}

%{
    private Symbol sym(int type)
    {
        return sym(type, yytext());
    }

    private Symbol sym(int type, Object value)
    {
        return new Symbol(type, yyline, yycolumn, value);
    }

    private void error()
    throws IOException
```

```
/* ANY          =    .
LineTerminator = \r | \n | \r\n
InputCharacter = [^\r\n]
WhiteSpace     = {LineTerminator} | [ \t\f]     /* The blank after the bracket is significant */



%%

/* {ANY}          {    return sym(ANY); }
[a-e] | [g-h] | [j-o] | [q-z] {return Symbol(Sym.ID);}
"f" {return Symbol.(Sym.FLTDCL);}
"i" {return Symbol.(Sym.INTDCL);}
"p" {return Symbol.(Sym.PRINT);}
"=" {return Symbol.(Sym.ASSIGN);}
"+" {return Symbol.(Sym.PLUS);}
"-" {return Symbol.(Sym.MINUS);}
([0-9])+ {return Symbol(Sym.INUM);}
([0-9])+"."([0-9])+ {return(Sym.FNUM);}

{WhiteSpace}                    { /* ignore */ }
```

# JLex generated Lexical Analyser

- Class Yylex
  - Name can be changed with %class directive
  - Default construction with one arg – the input stream
    - You can add your own constructors
  - The method performing lexical analysis is yylex()
    - Public Yytoken yylex() which return the next token
    - You can change the name of yylex() with %function directive
  - String yytext() returns the matched token string
  - Int yylength() returns the length of the token
  - Int yychar is the index of the first matched char (if %char used)
- Class Yytoken
  - Returned by yylex() – you declare it or supply one already defined
  - You can supply one with %type directive
    - Java_cup.runtime.Symbol is useful
  - Actions typically written to return Yytoken(…)

# Performance considerations

- Performance of scanners is important for production compilers, for example:
  - 30,000 lines per minute (500 lines per second)
  - 10,000 characters per second (for an average line of 20 characters)
  - For a processor that executes 10,000,000 instructions per second, 1,000 instructions per input character
  - Considering other tasks in compilers, 250 instructions per character is more realistic
- Size of scanner sometimes matters
  - Including keyword in scanner increases table size
    - E.g. Pascal has 35 keywords, including them increases states from 37 to 165
    - Uncompressed this increases table entries from 4699 to 20955
- Note modern scanners use explicit control, not table !
  - Why?

# Other Scanner Generators

- Flex:
  - It produces scanners than are faster than the ones produced by Lex
  - Options that allow tuning of the scanner size vs. speed
- JFlex: in Java
- GLA: Generator for Lexical Analyzers
  - It produces a directly executable scanner in C
  - It's typically twice as fast as Flex, and it's competitive with the best hand-written scanners
- re2c
  - It produces directly executable scanners
- Alex, Lexgen, …
- Others are parts of complete suites of compiler development tools
  - JavaCC
  - Coco/R
  - SableCC
  - ANTLR

# Conclusions

- Don't worry too much about DFAs
- You **do** need to understand how to specify regular expressions
- Note that different tools have different notations for regular expressions.
- You would probably only need to use Lex/Flex resp. Jlex/JFLex if you also use Yacc resp. CUP

- Sometimes it is easier to develop the scanner by hand transforming the RE into a case based direct scanner !
- In your project you can define the token grammar and implement a scanner by hand and/or by JFlex