Languages and Compilers (SProg og Oversættere)

Bent Thomsen
Department of Computer Science
Aalborg University

With acknowledgement to Norm Hutchinson whose slides this lecture is based on.

Programming Language Genealogy

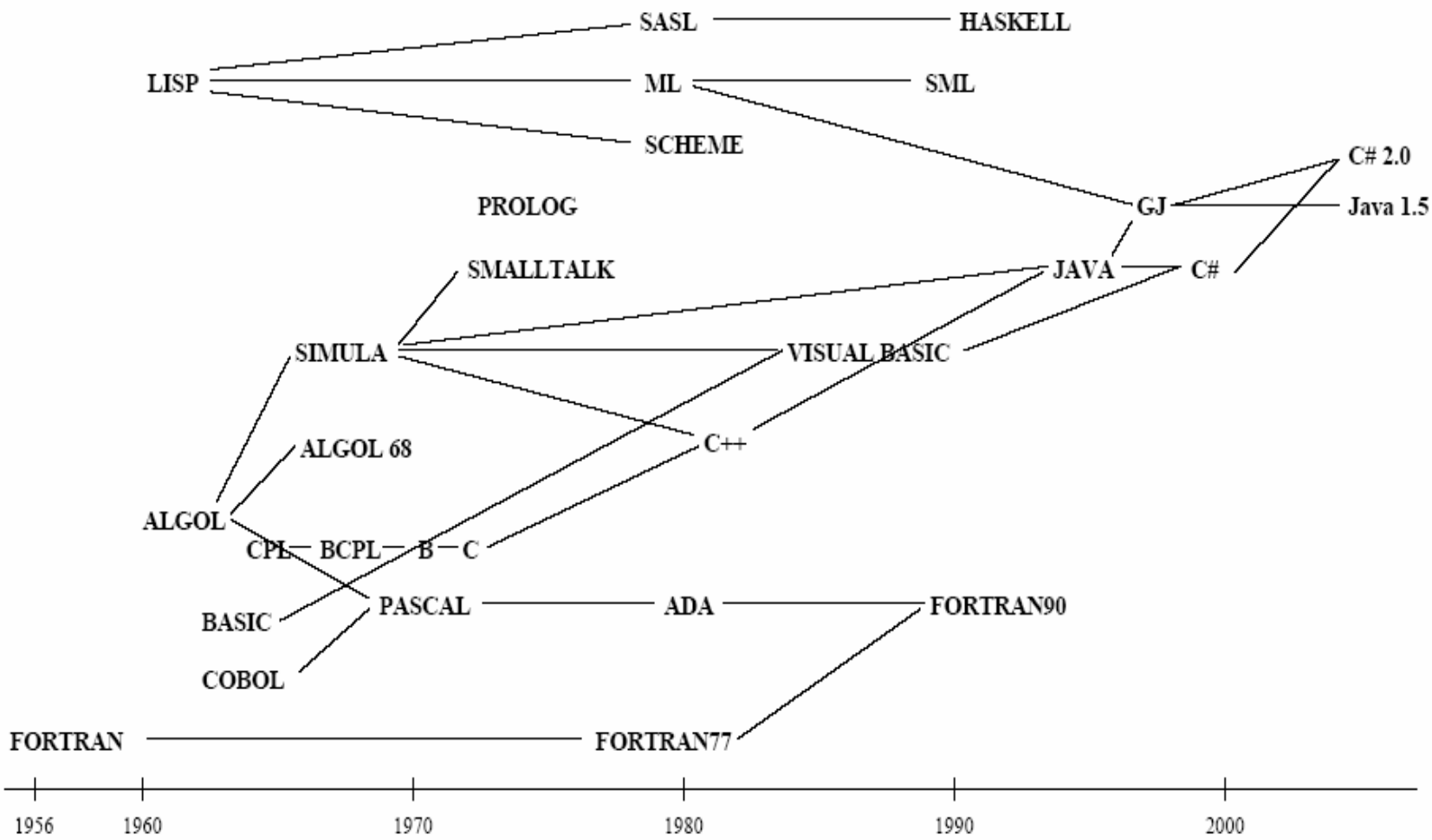


diagram.pdf Lang History.htm Diagram by Peter Sestoft

Offentliggjort 24. februar 2006 14:01 - opdateret 14:06 Tip en ven Print-version

Fornem IT-pris til dansker

Som den første dansker nogensinde tildeles Peter Naur, professor emeritus ved Københavns Universitet, ACM's Turing Award - også kaldet datalogiens svar på en Nobelpris.

Prisen er datalogiens højeste udmærkelse og uddeles en gang om året til personer, som har ydet et afgørende og varigt bidrag til området. Den ledsages af et beløb på 100.000 dollars, svarende til ca. 620.000 kroner, som er skænket af virksomheden Intel. Prisen vil blive overrakt ved en banket den 20. maj i San Francisco.

Ifølge ACM's priskomité har Peter Naur fået prisen for "grundlæggende bidrag til udformningen af programmeringssprog og definitionen af Algol 60, til udformningen af oversættere og til det kreative og praktiske arbejde med programmering".

/ritzau/

Tilbage til forsiden Til toppen af siden Tip en ven Print-version

Quick review

- Syntactic analysis
  - Prepare the grammar
    - Grammar transformations
      - Left-factoring
      - Left-recursion removal
      - Substitution
  - (Lexical analysis)
    - This lecture
  - Parsing - Phrase structure analysis
    - Group words into sentences, paragraphs and complete programs
    - Top-Down and Bottom-Up
    - Recursive Decent Parser
    - Construction of AST

Note: You will need (at least) two grammars

- One for Humans to read and understand
- (may be ambiguous, left recursive, have more productions than necessary, …)
- One for constructing the parser

A good question!

- Why do you tell us how we can do syntax checking when Eclipse (NetBeans or VisualStudio) does for us?

Language Processors: What are they?

A programming language processor is any system that manipulates programs.

Examples:

- Editors
  - Emacs
- Integrated Development Environments
  - Borland jBuilder
  - NetBeans
  - Eclipse
  - Visual Studio .Net
- Translators (e.g. compiler, assembler, disassembler)
- Interpreters

How does Eclipse know about Java Syntax?

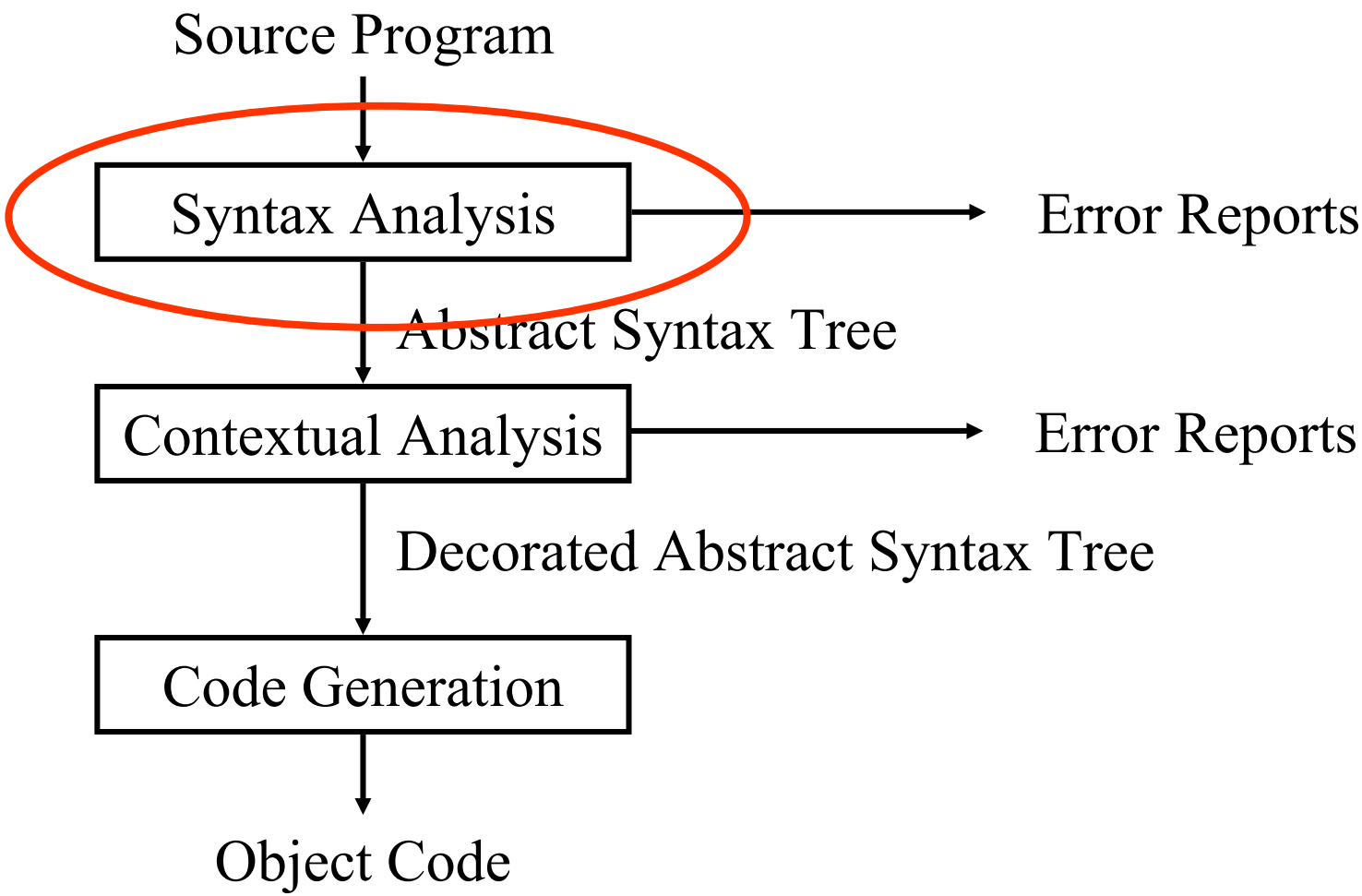
JDT Core

JDT Core is the Java infrastructure of the Java IDE. It includes:

- An incremental Java compiler. Implemented as an Eclipse builder, it is based on technology evolved from VisualAge for Java compiler. In particular, it allows to run and debug code which still contains unresolved errors.
- A Java Model that provides API for navigating the Java element tree. The Java element tree defines a Java centric view of a project. It surfaces elements like package fragments, compilation units, binary classes, types, methods, fields.
- A Java Document Model providing API for manipulating a structured Java source document.
- Code assist and code select support.
- An indexed based search infrastructure that is used for searching, code assist, type hierarchy computation, and refactoring. The Java search engine can accurately find precise matches either in sources or binaries.
- Evaluation support either in a scrapbook page or a debugger context.
- Source code formatter

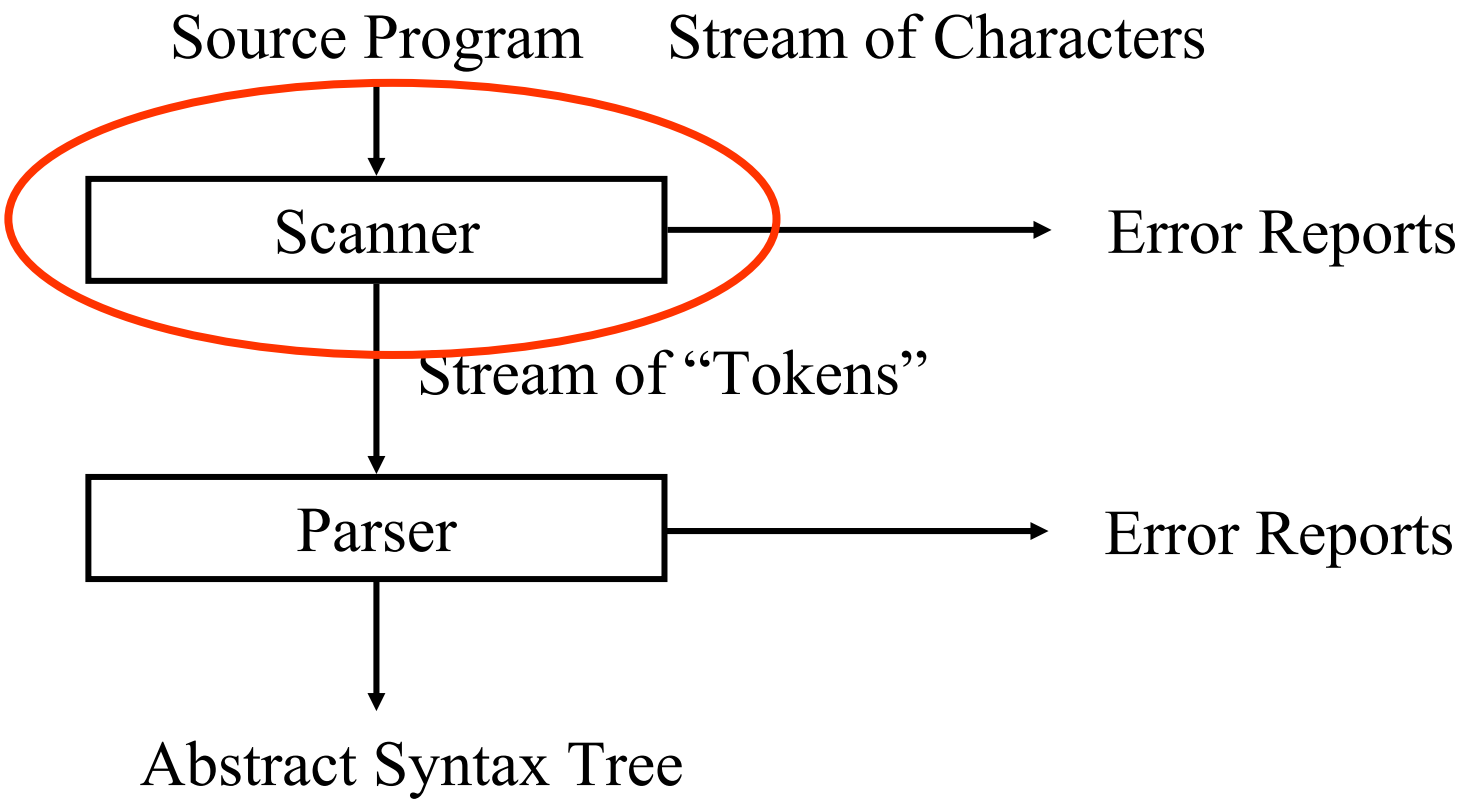
The JDT Core infrastructure has no built-in JDK version dependencies, it also does not depend on any particular Java UI and can be run headless.

The “Phases” of a Compiler



Syntax Analysis: Scanner

Dataflow chart

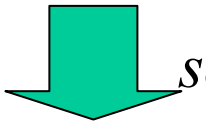


1) Scan: Divide Input into Tokens

An example mini Triangle source program:

```
let var y: Integer
in  !new year
    y := y+1
```

Tokens are "words" in the input, for example keywords, operators, identifiers, literals, etc.



let	*var*	*ident.*	*colon*	*ident.*	*in*
`let`	`var`	`y`	`:`	`Integer`	`in`

...

... | *ident.* | *becomes* | *ident.* | *op.* | *intlit* | *eot* |
|---|---|---|---|---|---|
| `y` | `:=` | `y` | `+` | `1` | |

Developing RD Parser for Mini Triangle

Last Lecture we just said:

- The following non-terminals are recognized by the scanner
- They will be returned as tokens by the scanner

```
Identifier := Letter (Letter|Digit)*
Integer-Literal ::= Digit Digit*
Operator ::= + | - | * | / | < | > | =
Comment ::= ! Graphic* eol
```

Assume scanner produces instances of:

```
public class Token {
   byte kind; String spelling;
   final static byte
      IDENTIFIER = 0,
      INTLITERAL = 1;
      ...
```

And this is where we need it

```
public class Parser {
   private Token currentToken;
   private void accept(byte expectedKind) {
      if (currentToken.kind == expectedKind)
         currentToken = scanner.scan();
      else
         report syntax error
   }
   private void acceptIt() {
      currentToken = scanner.scan();
   }
   public void parse() {
      acceptIt(); //Get the first token
      parseProgram();
      if (currentToken.kind != Token.EOT)
         report syntax error
   }
   ...
```

Steps for Developing a Scanner

1) Express the “lexical” grammar in EBNF (do necessary transformations)
2) Implement Scanner based on this grammar (details explained later)
3) Refine scanner to keep track of spelling and kind of currently scanned token.

To save some time we’ll do step 2 and 3 at once this time

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
```

Developing a Scanner

Implementation of the scanner

```
public class Scanner {

   private char currentChar;
   private StringBuffer currentSpelling;
   private byte currentKind;

   private char take(char expectedChar) { ... }
   private char takeIt() { ... }

   // other private auxiliary methods and scanning
   // methods here.

   public Token scan() { ... }
}
```

Developing a Scanner

The scanner will 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

```
public class Scanner {

  private char currentChar = get first source char;
  private StringBuffer currentSpelling;
  private byte currentKind;

  private char take(char expectedChar) {
    if (currentChar == expectedChar) {
      currentSpelling.append(currentChar);
      currentChar = get next source char;
    }
    else report lexical error
  }
  private char takeIt() {
    currentSpelling.append(currentChar);
    currentChar = get next source char;
  }
  ...
```

Developing a Scanner

```
...
public Token scan() {
   // Get rid of potential separators before
   // scanning a token
   while (  (currentChar == '!')
         || (currentChar == ' ')
         || (currentChar == '\n' ) )
      scanSeparator();
   currentSpelling = new StringBuffer();
   currentKind = scanToken();
   return new Token(currentkind,
                    currentSpelling.toString());
}

private void scanSeparator() { ... }
private byte scanToken() { ... }
...
```

Developed much in the same way as parsing methods

```
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.

But before we look at doing that, we need some theory!

Developing a Scanner

The scanner will 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;

...
}
```

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))

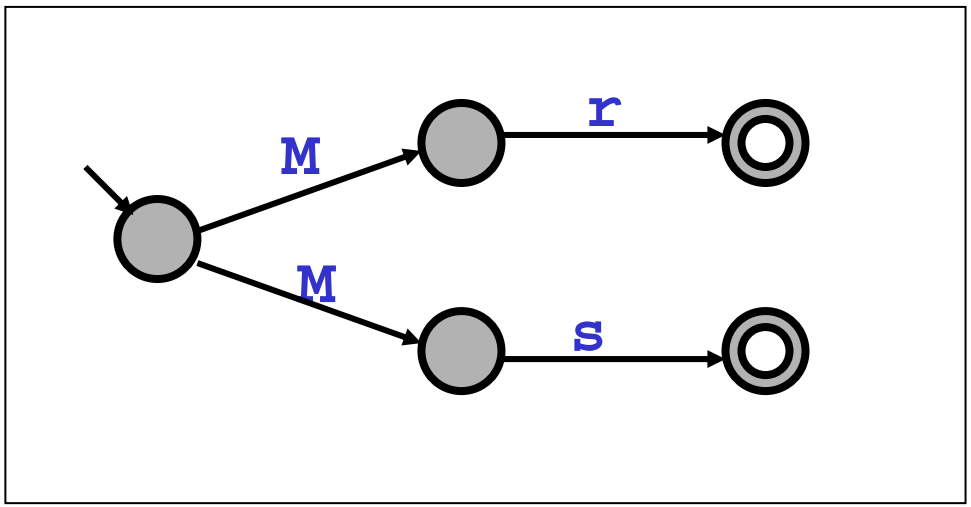
Definition: A finite state machine is an N-tuple $(States, \Sigma, start, \delta, End)$

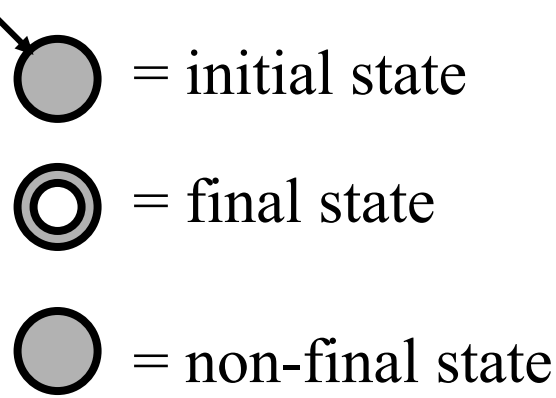
States	A finite set of "states"
Σ	An "alphabet": a finite set of symbols from which the strings we want to recognize are formed (for example: the ASCII char set)
start	A "start state" $Start \in States$
δ	Transition relation $\delta \subseteq States \times States \times \Sigma$. These are "arrows" between 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 is by means of a picture:

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

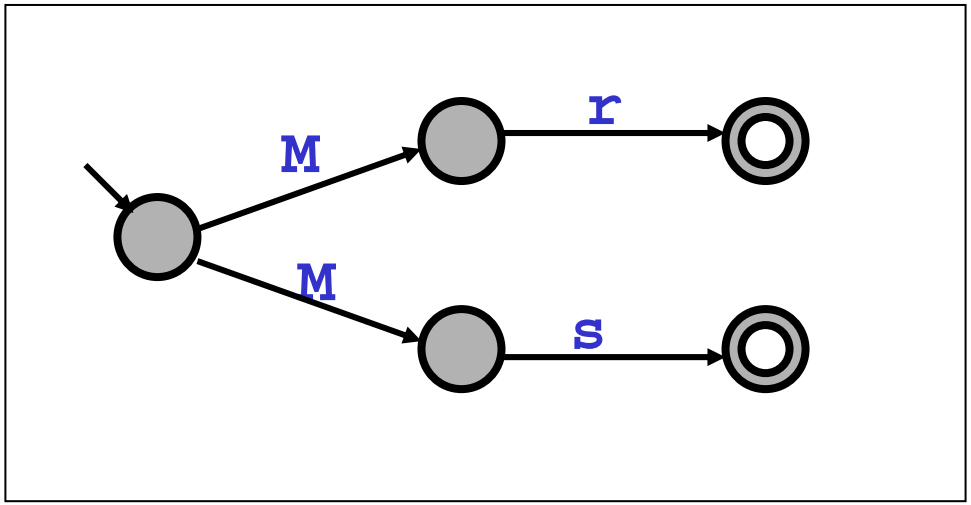




Deterministic, and non-deterministic DFA

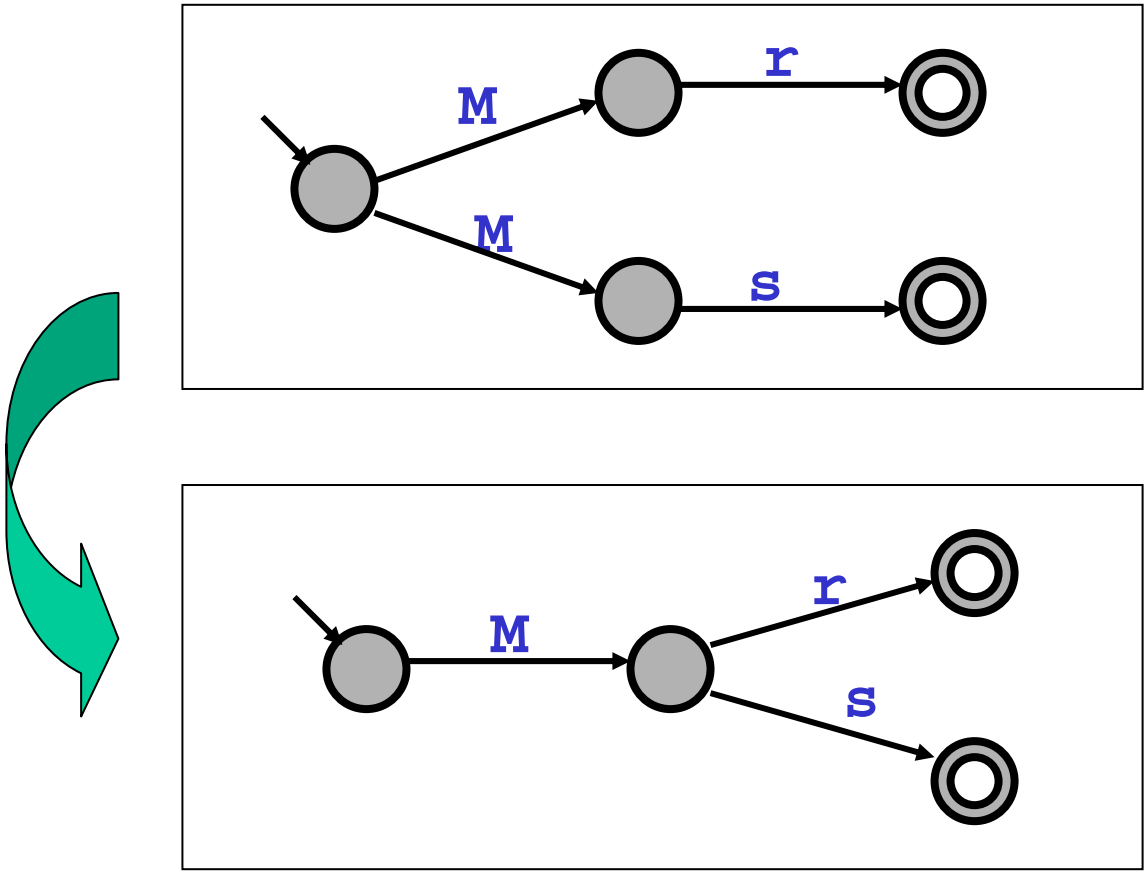
- A FA is called deterministic (acronym: DFA) if for every state and every possible input symbol, there is only one possible transition to chose from. Otherwise it is called non-deterministic (NDFA).

Q: Is this FSM deterministic or non-deterministic:



Deterministic, and non-deterministic FA

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



Deterministic, and non-deterministic FA

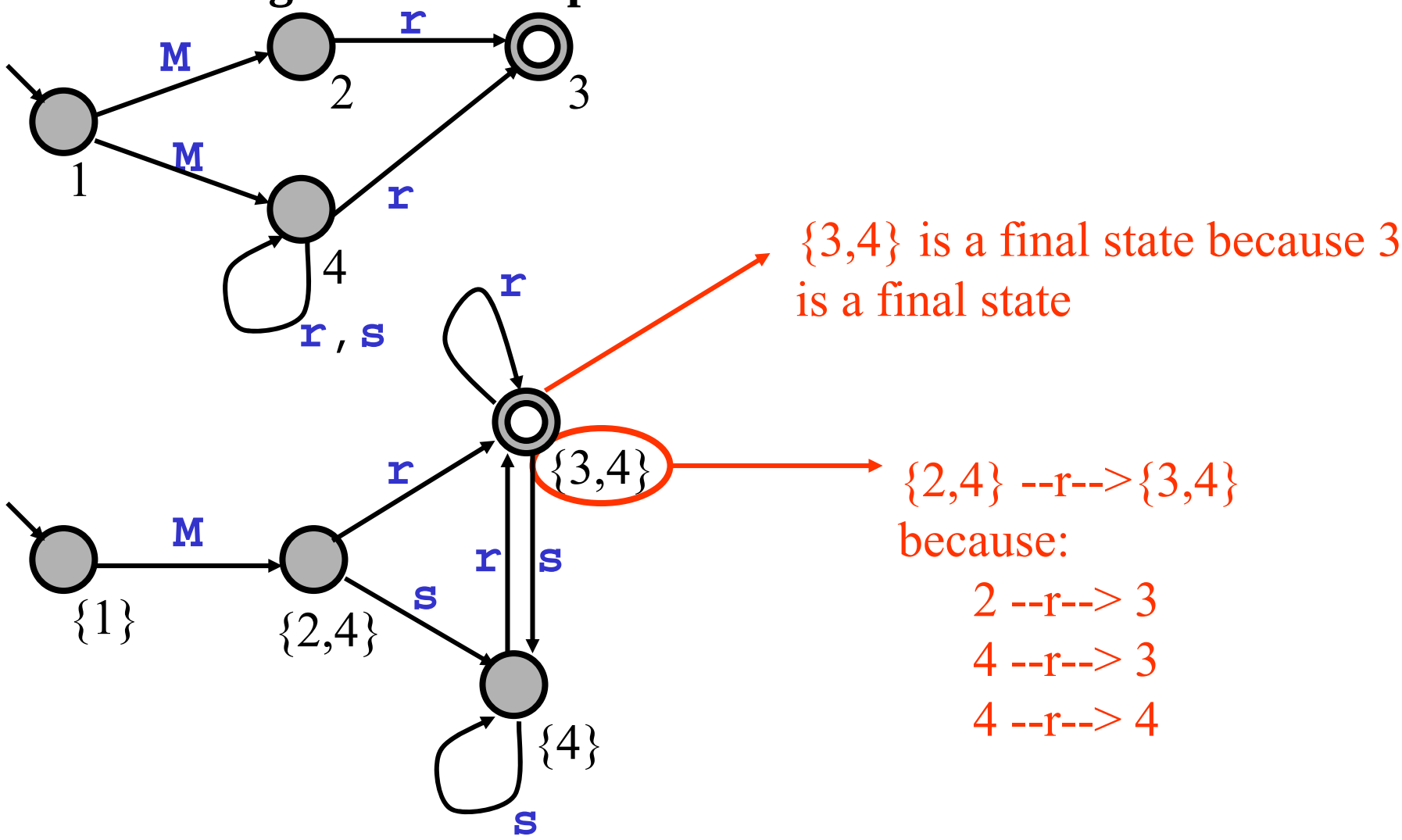
- Theorem: every NDFA can be converted into an equivalent DFA.

Algorithm:
The basic idea: DFA is defined as a machine that does a "parallel simulation" of the NDFA.

- The states of the DFA are subsets of the states of the NDFA (i.e. every state of the DFA is a set of states of the NDFA)
  => This state can be interpreted as meaning "the simulated DFA is now in any of these states"

Deterministic, and non-deterministic FA

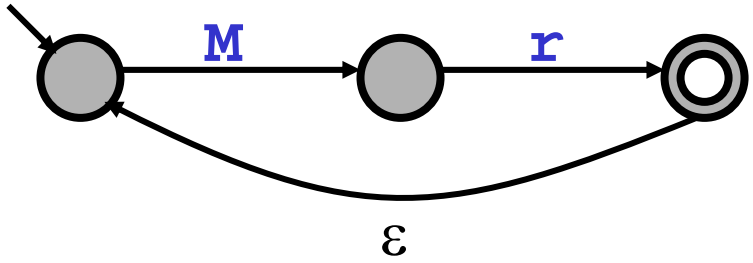
Conversion algorithm example:



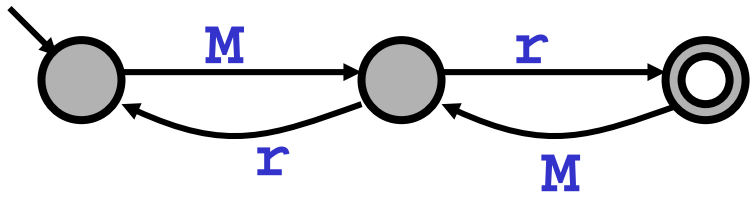
FA with ε moves

(N)DFA-ε automata are like (N)DFA. In an (N)DFA-ε we are allowed to have transitions which are "ε-moves".

Example: `M r (M r)*`



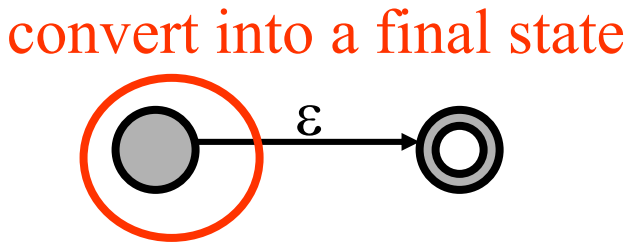
Theorem: every (N)DFA-ε can be converted into an equivalent NDFA (without ε-moves).



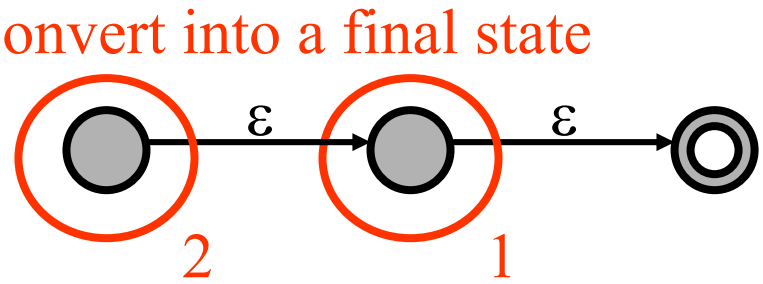
FA with ε moves

Theorem: every (N)DFA-ε can be converted into an equivalent NDFA (without ε-moves).

Algorithm:
1) converting states into final states:
if a final state can be reached from
a state S using an ε-transition
convert it into a final state.



Repeat this rule until no more states can be converted.
For example:



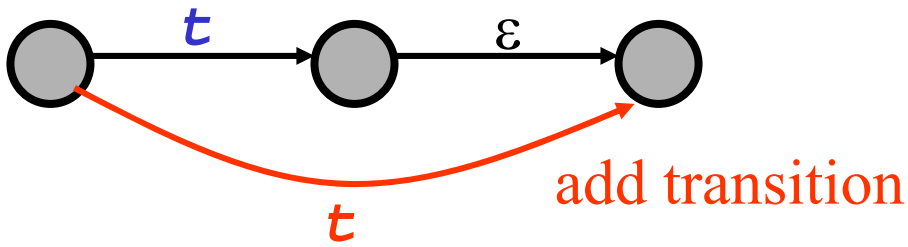
FA with ε moves

Algorithm:

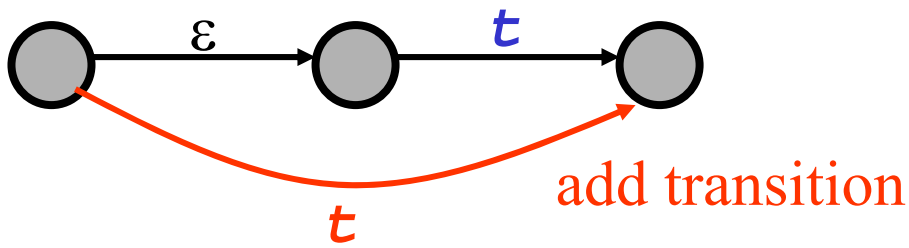
1) converting states into final states.

2) adding transitions (repeat until no more can be added)

a) for every transition **followed** by ε-transition



b) for every transition **preceded** by ε-transition

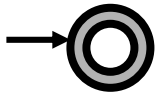


3) delete all ε-transitions

Converting a RE into an NDFA-ε

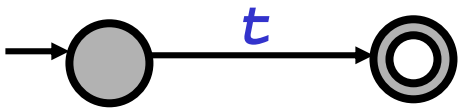
RE: ε

FA:



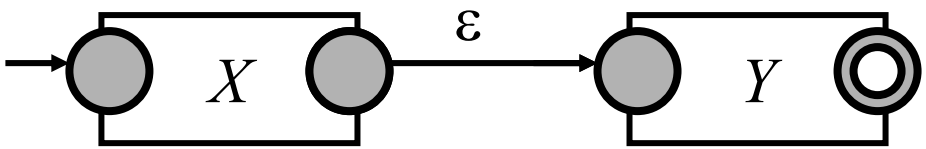
RE: `t`

FA:



RE: *XY*

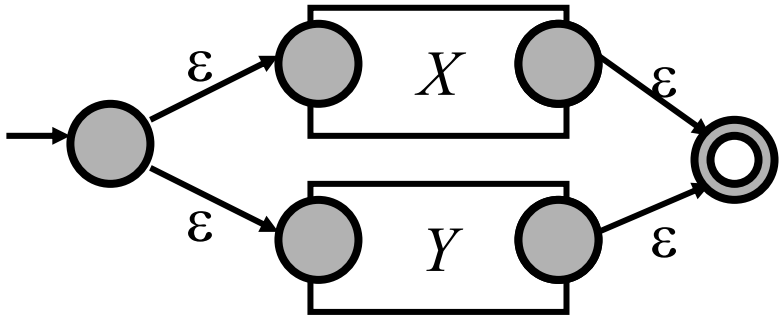
FA:



Converting a RE into an NDFA-ε

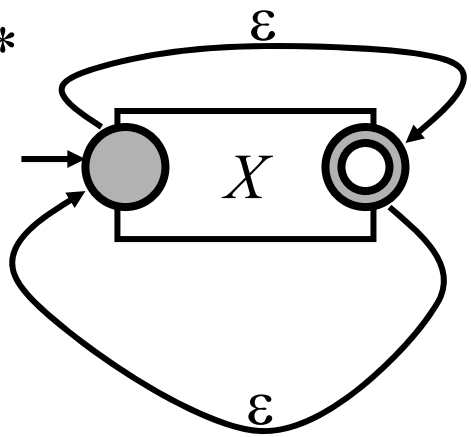
RE: $X|Y$

FA:



RE: X^*

FA:



FA and the implementation of Scanners

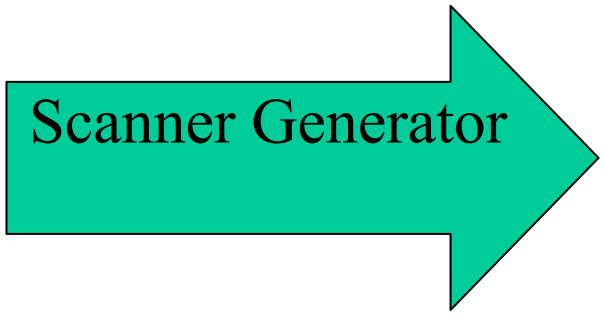
- Regular expressions, (N)DFA-ε and NDFA and DFA's are all equivalent formalism in terms of what languages can be defined with them.
- Regular expressions are a convenient notation for describing the "tokens" of programming languages.
- Regular expressions can be converted into FA's (the algorithm for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

will explain this in subsequent slides

FA and the implementation of Scanners

What a typical scanner generator does:

Token definitions
Regular expressions



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

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
Σ	**`byte`** or **`char`** data type.
start	An integer number
δ	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)

Implementing a DFA

```
public class Recognizer {
  static boolean[] finalState = final state table ;
  static int[][] delta = transition table ;

  private byte currentCharCode = get first char ;
  private int  currentState    = start state ;

  public boolean recognize() {
    while (currentCharCode is not end of file) &&
          (currentState is not error state ) {
       currentState =
           delta[currentState][currentCharCode];
       currentCharCode = get next char ;
    return finalState[currentState];
  }

}
```

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?

  Match longest possible token before reaching error state.

- How to identify matched token class (not just true|false)

  Final state determines matched token class

Implementing a Scanner as a DFA

```
public class Scanner {
  static int[] matchedToken = maps state to token class
  static int[][] delta = transition table ;

  private byte currentCharCode = get first char ;
  private int  currentState    = start state ;
  private int  tokbegin    = begining of current token ;
  private int  tokend      = end of current token
  private int tokenKind ;

  ...
```

Implementing a Scanner as a DFA

```
public Token scan() {
  skip separator (implemented as DFA as well)
  tokbegin = current source position
  tokenKind = error code
  while (currentState is not error state ) {
     if (currentState is final state ) {
         tokend = current source location ;
         tokenKind = matchedToken[currentState];
     currentState =
         delta[currentState][currentCharCode];
     currentCharCode = get next source char ;
  }
  if (tokenKind == error code ) report lexical error ;
  move current source position to tokend
  return new Token(tokenKind,
       source chars from tokbegin to tokend-1 );
}
```

We don’t do this by hand anymore!

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

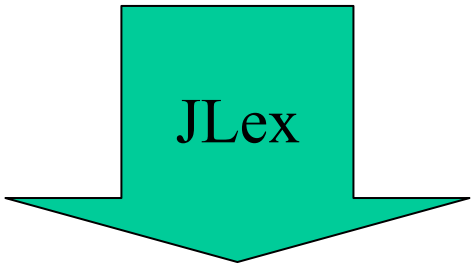
- JLex (JFlex)
  - input:
    - a set of REs and action code
  - output
    - a fast lexical analyzer (scanner)
      - based on a DFA

- Or the lexer is built into the parser generator as in JavaCC

JLex Lexical Analyzer Generator for Java

Definition of tokens

Regular Expressions



Java File: Scanner Class

Recognizes Tokens

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 JLex 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).
- The following characters are *metacharacters*.

  ? * + | ( ) ^ $. [ ] { } “ \

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

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`.

JLex Regular Expressions

- Parentheses are used for grouping.

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

- If a regular expression begins with ^, then it is matched only at the beginning of a line.
- If a regular expression ends with $, then it is matched only at the end of a line.
- The dot `.` matches any non-newline character.

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 nonletter.

JLex Regular Expressions

- A single character within double quotes " " represents itself.
- Metacharacters lose their special meaning and represent themselves when they stand alone within single quotes.
  - "?" matches ?.

JLex Escape Sequences

- 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`.

The JLex tool: Example

An example:

```
import java_cup.runtime.*;

%%

%class Lexer
%unicode
%cup
%line
%column
%state STRING

%{
...
```

The JLex tool

```
%state STRING

%{
  StringBuffer string = new StringBuffer();

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

%}


...
```

The JLex tool

```
%}
LineTerminator = \r|\n|\r\n
InputCharacter = [^\r\n]
WhiteSpace     = {LineTerminator} | [ \t\f]

/* comments */
Comment = {TraditionalComment} | {EndOfLineComment} |

TraditionalComment = "/*" {CommentContent} "*"+ "/"
EndOfLineComment= "//"{InputCharacter}* {LineTerminator}
CommentContent       = ( [^*] | \*+ [^/*] )*

Identifier = [:jletter:] [:jletterdigit:]*
DecIntegerLiteral = 0 | [1-9][0-9]*

%%
...
```

The JLex tool

```
...
%%

<YYINITIAL> "abstract"   { return symbol(sym.ABSTRACT); }
<YYINITIAL> "boolean"    { return symbol(sym.BOOLEAN); }

<YYINITIAL> "break"      { return symbol(sym.BREAK); }


<YYINITIAL> {
  /* identifiers */
  {Identifier}        { return symbol(sym.IDENTIFIER);}

  /* literals */
  {DecIntegerLiteral} { return symbol(sym.INT_LITERAL);}
  ...
```

The JLex tool

```
...
/* literals */
{DecIntegerLiteral} { return symbol(sym.INT_LITERAL);}
\"                  { string.setLength(0);
                      yybegin(STRING); }

/* operators */
"="                 { return symbol(sym.EQ); }
"=="                 { return symbol(sym.EQEQ); }
"+"                 { return symbol(sym.PLUS); }

/* comments */
{Comment}           { /* ignore */ }

/* whitespace */
{WhiteSpace}        { /* ignore */ }
}
...
```

The JLex tool

```
...
<STRING> {
  \"                { yybegin(YYINITIAL);
                      return symbol(sym.STRINGLITERAL,
                                    string.toString()); }
  [^\n\r\"\]+       { string.append( yytext() ); }
  \\t               { string.append('\t'); }
  \\n               { string.append('\n'); }

  \\r               { string.append('\r'); }
  \\"               { string.append('\"'); }
  \\                { string.append('\'); }

}
```

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 yylenght() 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(…)

Java.io.StreamTokenizer

- An alternative to JLex is to use the class *StreamTokenizer* from *java.io*
- The class recognizes 4 types of lexical elements (tokens):
  - number (sequence of decimal numbers eventually starting with the –(minus) sign and/or containing the decimal point)
  - word (sequence of characters and digits starting with a character)
  - line separator
  - end of file

Java.io.StreamTokenizer

StreamTokenizer tokens = new StreamTokenizer(input File);

nextToken() method move a tokenizer to the next token

token_variable.nextToken()

nextToken() returns the token type as its value

StreamTokenizer.TT_EOF : end-of-file reached
StreamTokenizer.TT_NUMBER : a number was scanned;the value is saved in nval(double); if it is an integer, it needs to be typecasted into int ((int)tokens.nval)
StreamTokenizer.TT_WORD : a word was scanned; the value is saved in sval(String)

Java.io.StreamTokenizer

```
public class AltScanner extends Scanner {

  StreamTokenizer st;
  private StringBuffer currentSpelling;
  private boolean debug = false;
  public AltScanner(){};
  /** Creates a new instance of AltScanner */
  public AltScanner(StreamTokenizer ist) {
    st = ist;
    // process the entire file, of space or comma-delimited ints
    // Prepare the tokenizer for Java-style tokenizing rules
    st.parseNumbers();
    //st.wordChars('_', '_');
    //st.eolIsSignificant(true);

    // If whitespace is not to be discarded, make this call
    //st.ordinaryChars(0, ' ');

    // These calls caused comments to be discarded
    //st.slashSlashComments(true);
    //st.slashStarComments(true);
    st.commentChar('!');
  }

  public void enableDebugging() {
  debug = true;
```

```
public Token scan () {
Token tok;
SourcePosition pos;
int kind;
int intoken;

currentSpelling = new StringBuffer("");
pos = new SourcePosition();
pos.start = st.lineno();

try
{
  intoken = st.nextToken();
  if (st.sval != null) currentSpelling.append(st.sval);
  else currentSpelling.append((char)st.ttype);
  switch(intoken){
  case '+': case '-': case '*': case '/': case '=':
  case '<': case '>': case '\\': case '&': case '@':
  case '%': case '^': case '?':
    //needs to read next symbol as well
    kind = Token.OPERATOR;
    //if (isOperator(st.nextToken())){};
    if (st.nextToken() == '=') {
      currentSpelling.append((char)st.ttype);
   } else {
    st.pushBack();
   };
    break;
  case '\'':
    String squoteVal = st.sval;
    kind = Token.CHARLITERAL;
    break;
```

```
    break;
  case ']':
    kind = Token.RBRACKET;
    break;
  case '{':
    kind = Token.LCURLY;
    break;
  case '}':
    kind = Token.RCURLY;
    break;
  case SourceFile.EOT:
    kind = Token.EOT;
    break;
  case StreamTokenizer.TT_WORD:
  // A word was found; the value is in sval
      kind = Token.IDENTIFIER;
      break;
    case StreamTokenizer.TT_NUMBER:
      int num = (int)st.nval;
      currentSpelling = new StringBuffer(""+num);
      kind = Token.INTLITERAL;
      break;
    case StreamTokenizer.TT_EOF:
      // End of file has been reached
      kind = Token.EOT;
      break;
    default:
      // A regular character was found; the value is the token itself
      kind = Token.ERROR;
      break;
  }
  }
  catch (IOException e)
  {
    kind = Token.ERROR;
  }
```

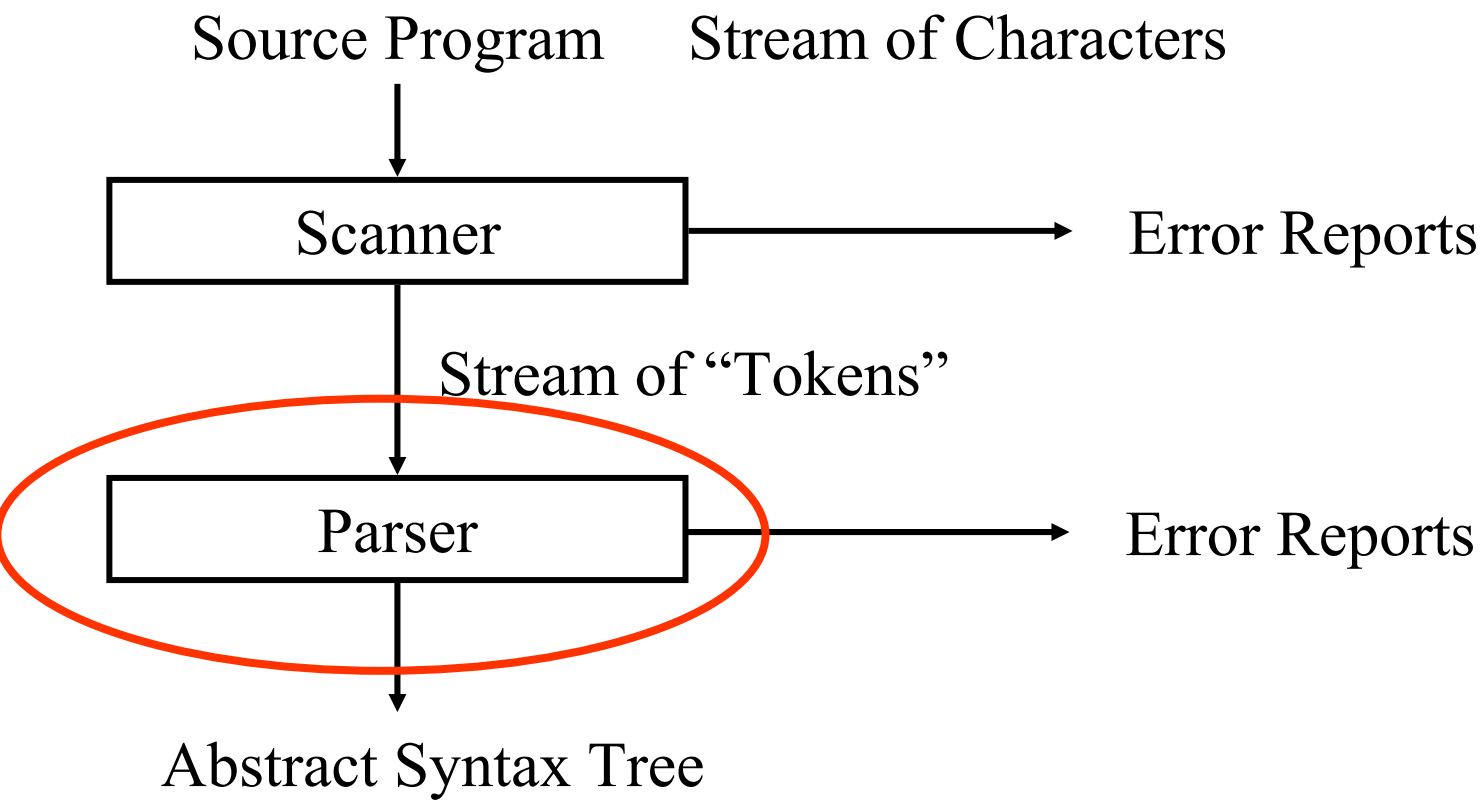
```
        default:
          // A regular character was found; the value is the token itself
          kind = Token.ERROR;
          break;
      }
      }
      catch (IOException e)
      {
        kind = Token.ERROR;
      }


//kind = scanToken();

      pos.finish = st.lineno();
      tok = new Token(kind, currentSpelling.toString(), pos);
      if (debug)
        System.out.println(tok);
      return tok;
      }

}
```

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 JLex (Lex) if you use also use CUP (or Yacc or SML-Yacc)

Syntax Analysis: Parser

Dataflow chart



Systematic Development of RD Parser

(1) Express grammar in EBNF

(2) Grammar Transformations:

Left factorization and Left recursion elimination

(3) Create a parser class with

- private variable `currentToken`
- methods to call the scanner: `accept` and `acceptIt`

(4) Implement private parsing methods:

- add private `parse`***N*** method for each non terminal ***N***
- public `parse` method that
  - gets the first token form the scanner
  - calls `parse`***S*** (***S*** is the start symbol of the grammar)

Algorithm to convert EBNF into a RD parser

- The conversion of an EBNF specification into a Java implementation for a recursive descent parser is straightforward
- We can describe the algorithm by a set of mechanical rewrite rules

N ::= *X*

```
private void parseN() {
   parse X
}
```

Algorithm to convert EBNF into a RD parser

parse t where *t* is a terminal

accept(*t*);

parse N where *N* is a non-terminal

parse*N*();

parse ε

// a dummy statement

parse XY

parse X
parse Y

Algorithm to convert EBNF into a RD parser

*parse X**

```
while (currentToken.kind is in starters[X]) {
   parse X
}
```

parse X|Y

```
switch (currentToken.kind) {
   cases in starters[X]:
     parse X
     break;
   cases in starters[Y]:
     parse Y
     break;
   default: report syntax error
}
```

LL(1) Grammars

- The presented algorithm to convert EBNF into a parser does not work for all possible grammars.
- It only works for so called “LL(1)” grammars.
- What grammars are LL(1)?
- Basically, an **LL1 grammar** is a grammar which can be parsed with a **top-down parser** with a **lookahead** (in the input stream of tokens) of **one token**.

How can we recognize that a grammar is (or is not) LL1?

⇒We can deduce the necessary conditions from the parser generation algorithm.

⇒There is a formal definition we can use.

LL 1 Grammars

*parse X**

```
while (currentToken.kind is in starters[X]) {
   parse X
}
```

Condition: *starters*[*X*] must be disjoint from the set of tokens that can immediately follow *X* *

parse X|Y

```
switch (currentToken.kind) {
   cases in starters[X]:
     parse X
     break;
   cases in starters[Y]:
     parse Y
     break;
   default: report syntax error
}
```

Condition: *starters*[*X*] and *starters*[*Y*] must be disjoint sets.

Formal definition of LL(1)

A grammar G is LL(1) iff
for each set of productions M ::= $X_1 \mid X_2 \mid \ldots \mid X_n$:

1. *starters*[X_1], *starters*[X_2], …, *starters*[X_n] are all pairwise disjoint
2. *If* X_i =>* ε then *starters*[X_j]∩ *follow*[X]=Ø, for $1 \leq j \leq n. i \neq j$

If G is ε-free then 1 is sufficient

Derivation

- What does X_i =>* ε mean?
- It means a derivation from X_i leading to the empty production
- What is a derivation?
  - A grammar has a *derivation*:

  $\alpha A \beta$ => $\alpha \gamma \beta$ iff $A \rightarrow \gamma \in$ P (Sometimes A ::= γ)

  =>* is the transitive closure of =>
- Example:
  - G = ({*E*}, {*a*,+,*,(,)}, *P*, *E*)

  where $P = \{E \rightarrow E+E,\ E \rightarrow E*E,$

  $E \rightarrow a,\ E \rightarrow (E)\}$
  - *E* => *E*+*E* => *E*+*E***E* => *a*+*E***E* => *a*+*E***a* => *a*+*a***a*
  - *E* =>* *a*+*a***a*

Follow Sets

- Follow(A) is the set of prefixes of strings of terminals that can follow any derivation of A in G
  - \$ $\in$ follow(S) (sometimes <eof> $\in$ follow(S))
  - if ($B \rightarrow \alpha A \beta$) $\in P$, then
  - first(β)$\oplus$follow(B)$\subseteq$ follow(A)

- The definition of follow usually results in recursive set definitions. In order to solve them, you need to do several iterations on the equations.

A few provable facts about LL(1) grammars

- No left-recursive grammar is LL(1)
- No ambiguous grammar is LL(1)
- Some languages have no LL(1) grammar
- A ε-free grammar, where each alternative X_j for N ::= X_j begins with a distinct terminal, is a simple LL(1) grammar

Converting EBNF into RD parsers

- The conversion of an EBNF specification into a Java implementation for a recursive descent parser is so “mechanical” that it can easily be automated!

=> JavaCC “Java Compiler Compiler”

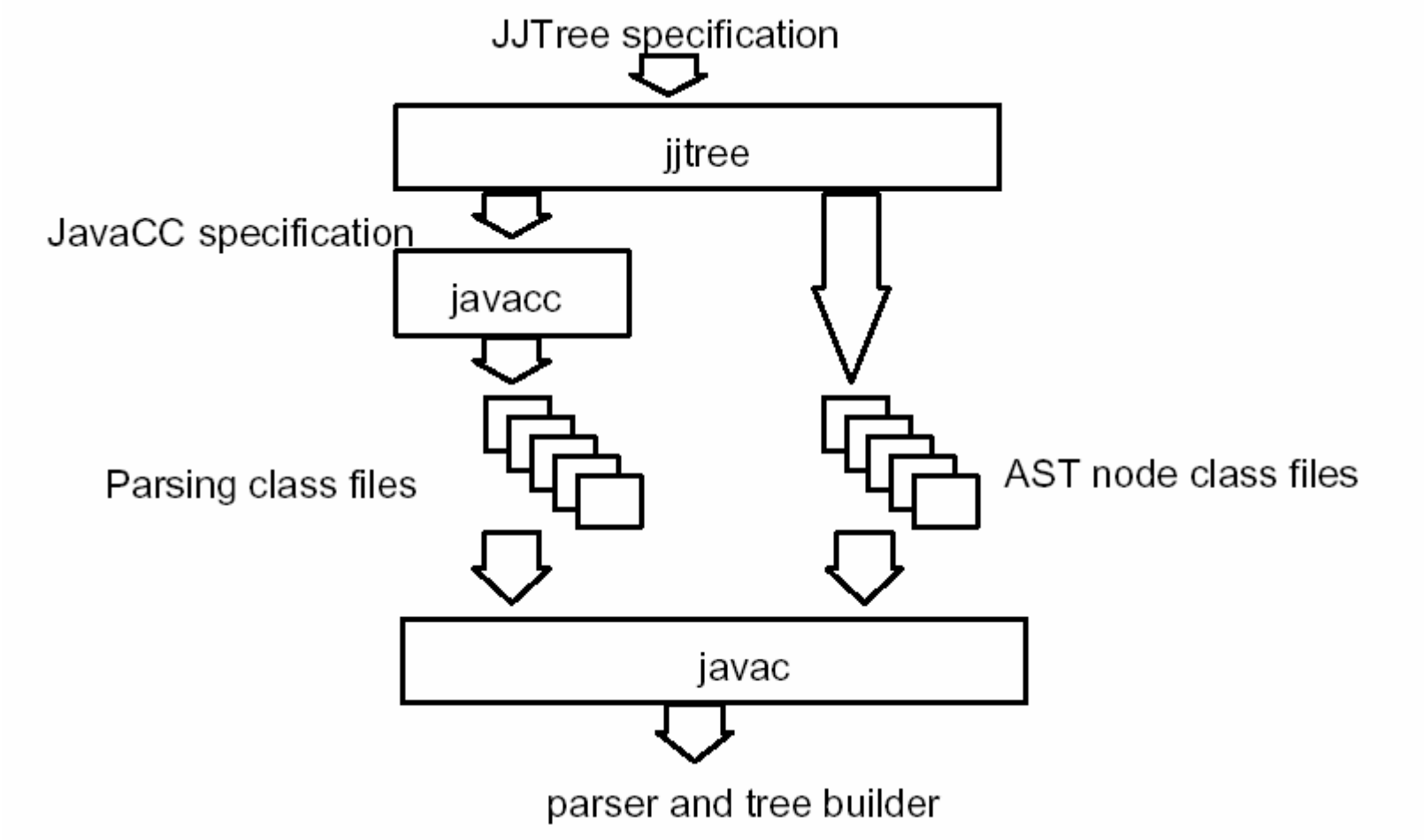
JavaCC

- JavaCC is a parser generator
- JavaCC can be thought of as “Lex and Yacc” for implementing parsers in Java
- JavaCC is based on LL(k) grammars
- JavaCC transforms an EBNF grammar into an LL(k) parser
- The lookahead can be change by writing LOOKAHEAD(…)
- The JavaCC can have action code written in Java embedded in the grammar
- JavaCC has a companion called JJTree which can be used to generate an abstract syntax tree

JavaCC and JJTree

- JavaCC is a parser generator
  - Inputs a set of token definitions, grammar and actions
  - Outputs a Java program which performs lexical analysis
    - Finding tokens
    - Parses the tokens according to the grammar
    - Executes actions
- JJTree is a preprocessor for JavaCC
  - Inputs a grammar file
  - Inserts tree building actions
  - Outputs JavaCC grammar file
- From this you can add code to traverse the tree to do static analysis, code generation or interpretation.

JavaCC and JJTree



JavaCC input format

- One file with extension .jj containing
  - Header
  - Token specifications
  - Grammar
- Example:

```
TOKEN:
{
        <INTEGER_LITERAL: (["1"-"9"](["0"-"9"])*|"0")>
}

void StatementListReturn() :
{}
{
        (Statement())* "return" Expression() ";"
}
```

JavaCC token specifications use regular expressions

- Characters and strings must be quoted
  - “;”, “int”, “while”
- Character lists […] is shorthand for |
  - [“a”-”z”] matches “a” | “b” | “c” | … | “z”
  - [“a”,”e”,”i”,”o”,u”] matches any vowel
  - [“a”-”z”,”A”-”Z”] matches any letter
- Repetition shorthand with * and +
  - [“a”-”z”,”A”-”Z”]* matches zero or more letters
  - [“a”-”z”,”A”-”Z”]+ matches one or more letters
- Shorthand with ? provides for optionals:
  - (“+”|”-”)?[“0”-”9”]+ matches signed and unsigned integers
- Tokens can be named
  - TOKEN : {<IDENTIFIER:<LETTER>(<LETTER>|<DIGIT>)*>}
  - TOKEN : {<LETTER: [“a”-”z”,”A”-”Z”] >|<DIGIT:[“0”-”9”]>}
  - Now <IDENTIFIER> can be used in defining syntax

A bigger example

```
options
{
   LOOKAHEAD=2;
}

PARSER_BEGIN(Arithmetic)

public class Arithmetic
{
}

PARSER_END(Arithmetic)

SKIP :
{
   " "
|  "\r"
|  "\t"
}

TOKEN:
{
   < NUMBER: (<DIGIT>)+ ( "." (<DIGIT>)+ )? >
|  < DIGIT: ["0"-"9"] >
}
```

```
double expr():
{
}
{
   term() ( "+" expr() | "-" expr() )*
}

double term():
{
}
{
   unary() ( "*" term() | "/" term() )*
}

double unary():
{
}
{
   "-" element() | element()
}

double element():
{
}
{
   <NUMBER> | "(" expr() ")"
}
```

Generating a parser with JavaCC

- Javacc *filename.jj*
  - generates a parser with specified name
  - Lots of .java files
- Javac *.java
  - Compile all the .java files

- Note the parser doesn't do anything on its own.
- You have to either
  - Add actions to grammar by hand
  - Use JJTree to generate actions for building AST
  - Use JBT to generate AST and visitors

Adding Actions by hand

```
options
{
   LOOKAHEAD=2;
}

PARSER_BEGIN(Calculator)

public class Calculator
{
   public static void main(String args[]) throws ParseException
   {
      Calculator parser = new Calculator(System.in);
      while (true)
      {
         parser.parseOneLine();
      }
   }
}

PARSER_END(Calculator)
```

```
SKIP :
{
   " "
|  "\r"
|  "\t"
}

TOKEN:
{
   < NUMBER: (<DIGIT>)+ ( "." (<DIGIT>)+ )? >
|  < DIGIT: ["0"-"9"] >
|  < EOL: "\n" >
}

void parseOneLine():
{
   double a;
}
{
   a=expr() <EOL>      { System.out.println(a); }
  | <EOL>
  | <EOF>              { System.exit(-1); }
}
```

Adding Actions by hand (ctd.)

```
double expr():
{
   double a;
   double b;
}
{
   a=term()
   (
      "+" b=expr()    { a += b; }
   |  "-" b=expr()    { a -= b; }
   )*
                  { return a; }
}

double term():
{
   double a;
   double b;
}
{
   a=unary()
   (
      "*" b=term()    { a *= b; }
   |  "/" b=term()    { a /= b; }
   )*
                  { return a; }
}
```

```
double unary():
{
   double a;
}
{
   "-" a=element()     { return -a; }
|  a=element()         { return a; }
}

double element():
{
   Token t;
   double a;
}
{
   t=<NUMBER>          { return Double.parseDouble(t.toString()); }
|  "(" a=expr() ")"     { return a; }
}
```

Using JJTree

- JJTree is a preprocessor for JavaCC
- JTree transforms a bare JavaCC grammar into a grammar with embedded Java code for building an AST
  - Classes Node and SimpleNode are generated
  - Can also generate classes for each type of node
- All AST nodes implement interface Node
  - Useful methods provided include:
    - Public void jjtGetNumChildren()
      - Which returns the number of children
    - Public void jjtGetChild(int i)
      - Which returns the I'th child
  - The "state" is in a parser field called jjtree
    - The root is at Node rootNode()
    - You can display the tree with
    - ((SimpleNode)parser.jjtree.rootNode()).dump(" ");
- JJTree supports the building of abstract syntax trees which can be traversed using visitors

JBT

- JBT – Java Tree Builder is an alternative to JJTree
- It takes a plain JavaCC grammar file as input and automatically generates the following:
  - A set of syntax tree classes based on the productions in the grammar, utilizing the Visitor design pattern.
  - Two interfaces: Visitor and ObjectVisitor. Two depth-first visitors: DepthFirstVisitor and ObjectDepthFirst, whose default methods simply visit the children of the current node.
  - A JavaCC grammar with the proper annotations to build the syntax tree during parsing.
- New visitors, which subclass DepthFirstVisitor or ObjectDepthFirst, can then override the default methods and perform various operations on and manipulate the generated syntax tree.

The Visitors Pattern

For object-oriented programming the *visitors pattern* enables the definition of a *new operator* on an *object structure* without *changing the classes* of the objects

When using visitor pattern

- The set of classes must be fixed in advanced
- Each class must have an accept method
- Each accept method takes a visitor as argument
- The purpose of the accept method is to invoke the visitor which can handle the current object.
- A visitor contains a visit method for each class (overloading)
- A method for class C takes an argument of type C

- The advantage of Visitors: New methods without recompilation!

Parser Generator Tools

- JavaCC is a Parser Generator Tool
- It can be thought of as Lex and Yacc for Java
- There are several other Parser Generator tools for Java
  - We shall look at some of them later

- Beware! To use Parser Generator Tools efficiently you need to understand what they do and what the code they produce does
- Note, there can be bugs in the tools and/or in the code they generate!

So what can I do with this in my project?

- Language Design
  - What type of language are you designing?
    - C like, Java Like, Scripting language, Functional ..
    - Does it fit with existing paradigms/generations?
    - Which language design criteria do emphasise?
      - Readability, Writability, orthogonality, …
  - What is the syntax?
    - Informal description
    - CFG in EBNF (Human readable, i.e. may be ambiguous)
    - LL(1) or (LALR(1)) grammar
    - Grammar transformations
      - left-factorization, left-recursion elimination, substitution
- Compiler Implementation
  - Recursive Decent Parser
  - (Tools generated Parser)