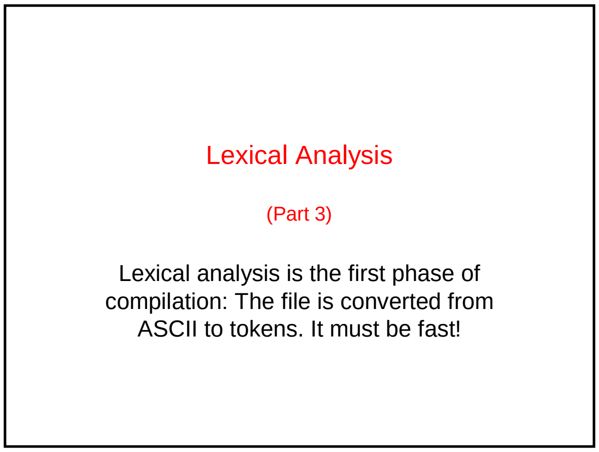
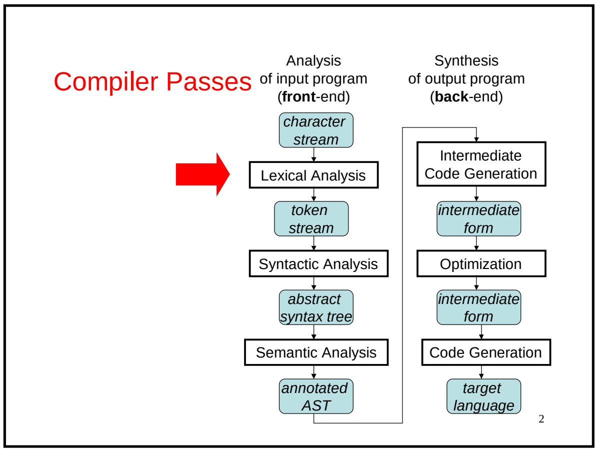


Lexical Analysis

(Part 3)

Lexical analysis is the first phase of compilation: The file is converted from ASCII to tokens. It must be fast!



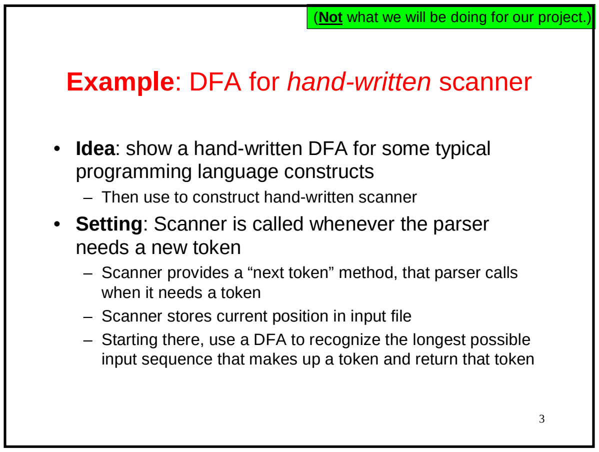
Compiler Passes

Analysis of input program (**front**-end)

Synthesis of output program (**back**-end)

character stream → Lexical Analysis → *token stream* → Syntactic Analysis → *abstract syntax tree* → Semantic Analysis → *annotated AST* → Intermediate Code Generation → *intermediate form* → Optimization → *intermediate form* → Code Generation → *target language*

2

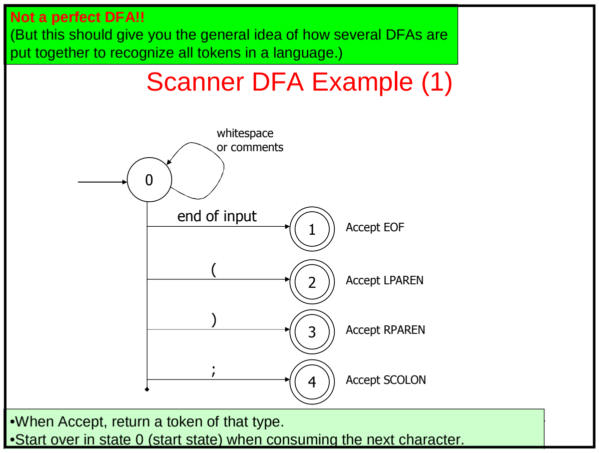


(**Not** what we will be doing for our project.)

Example: DFA for *hand-written* scanner

- **Idea**: show a hand-written DFA for some typical programming language constructs
  - Then use to construct hand-written scanner
- **Setting**: Scanner is called whenever the parser needs a new token
  - Scanner provides a "next token" method, that parser calls when it needs a token
  - Scanner stores current position in input file
  - Starting there, use a DFA to recognize the longest possible input sequence that makes up a token and return that token

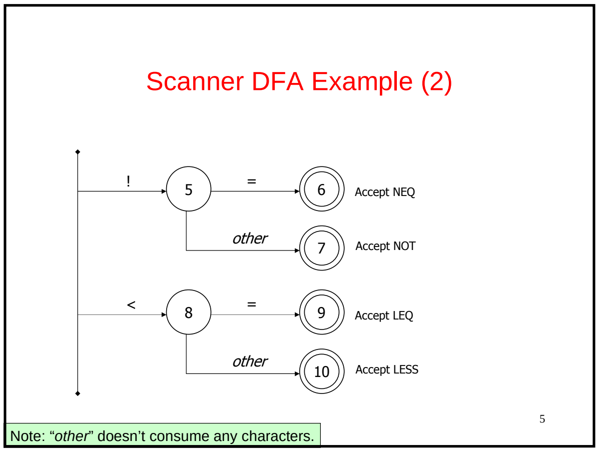
3



Not a perfect DFA!!
(But this should give you the general idea of how several DFAs are put together to recognize all tokens in a language.)

Scanner DFA Example (1)

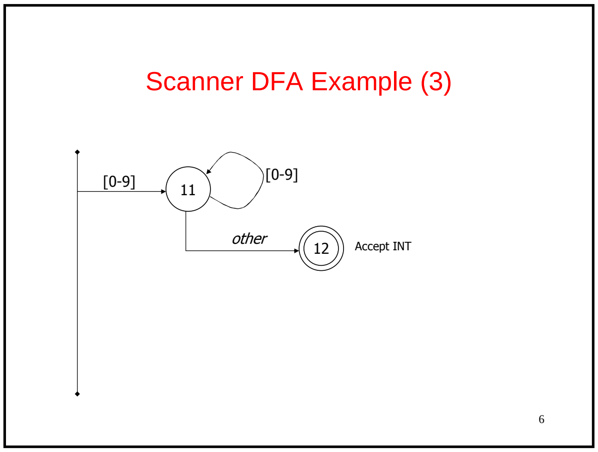
- When Accept, return a token of that type.
- Start over in state 0 (start state) when consuming the next character.



Scanner DFA Example (2)

5

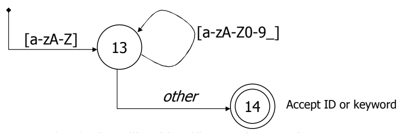
Note: "*other*" doesn't consume any characters.



Scanner DFA Example (3)

6

Scanner DFA Example (4)



- Strategies for handling identifiers vs keywords:
  - **Hand-written scanner**: look up identifier-like things in table of keywords to classify (good application of perfect hashing)
  - **Machine-generated scanner**: generate DFA with appropriate transitions to recognize keywords
    - Lots ’o states, but efficient (no extra lookup step)

DFA => Code

- Option 1: Implement by hand using procedures
  - (one procedure for each token)
  - (each procedure reads one character)
  - choices implemented using if and switch statements
- Pros
  - straightforward to write
  - fast
- Cons
  - a fair amount of tedious work
  - may have subtle differences from the language specification

(**Not** what we will be doing for our project.)

Implementing a Scanner by Hand: Scanner getToken() method

```
static char nextch;    // next unprocessed input character
void getch() { … }     // advance to next input char



public Token getToken() {     // return next input token
  Token result;

  skipWhiteSpace();

  if (no more input) {
   result = new Token(Token.EOF); return result;
  }

  switch(nextch) {
   case '(': result = new Token(Token.LPAREN); getch(); return result;
   case ‘)': result = new Token(Token.RPAREN); getch(); return result;
   case ‘;': result = new Token(Token.SCOLON); getch(); return result;

   // etc. …
```

Implementing a Scanner by Hand: getToken() (2)

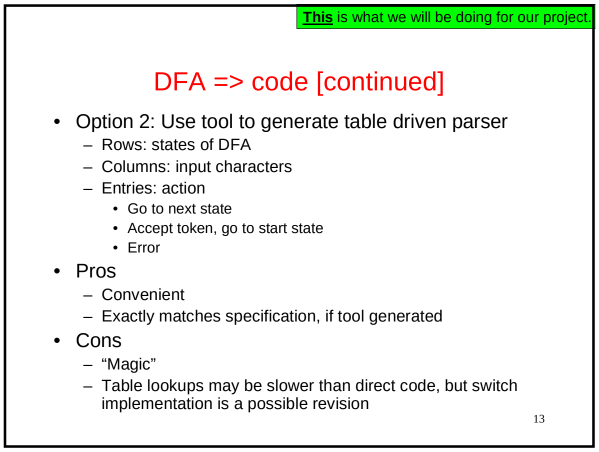
```
case '!': // ! or !=
        getch();
        if (nextch == '=') {
          result = new Token(Token.NEQ); getch(); return result;
        } else {
          result = new Token(Token.NOT); return result;
        }

case '<': // < or <=
        getch();
        if (nextch == '=') {
          result = new Token(Token.LEQ); getch(); return result;
        } else {
          result = new Token(Token.LESS); return result;
        }
// etc. …
```

Implementing a Scanner by Hand: getToken() (3)

```
case '0': case '1': case '2': case '3': case '4':
 case '5': case '6': case '7': case '8': case '9':
        // integer constant
        String num = nextch;
        getch();
        while (nextch is a digit) {
           num = num + nextch;
           getch();
        }
        result = new Token(Token.INT, Integer(num).intValue());
        return result;
// etc. …
```

Implementing a Scanner by Hand: getToken (4)

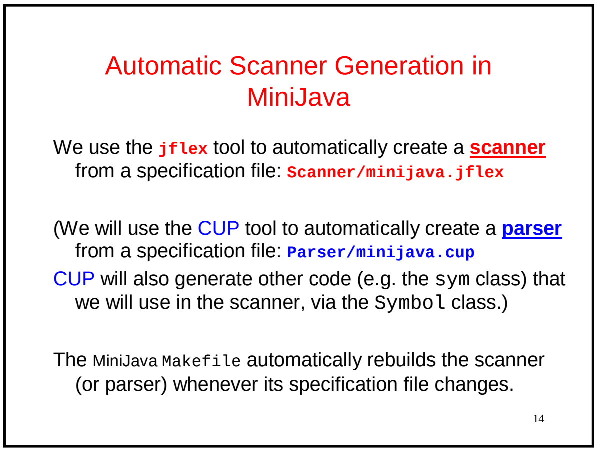
```
case 'a': … case 'z':
case 'A': … case 'Z':  // id or keyword
     string s = nextch;
     getch();
     while (nextch is a letter, digit, or underscore) {
        s = s + nextch;
        getch();
     }
     if (s is a keyword) {
        result = new Token(keywordTable.getKind(s));
     } else {
        result = new Token(Token.ID, s);
     }
     return result;
```



This is what we will be doing for our project.

DFA => code [continued]

- Option 2: Use tool to generate table driven parser
  - Rows: states of DFA
  - Columns: input characters
  - Entries: action
    - Go to next state
    - Accept token, go to start state
    - Error
- Pros
  - Convenient
  - Exactly matches specification, if tool generated
- Cons
  - “Magic”
  - Table lookups may be slower than direct code, but switch implementation is a possible revision



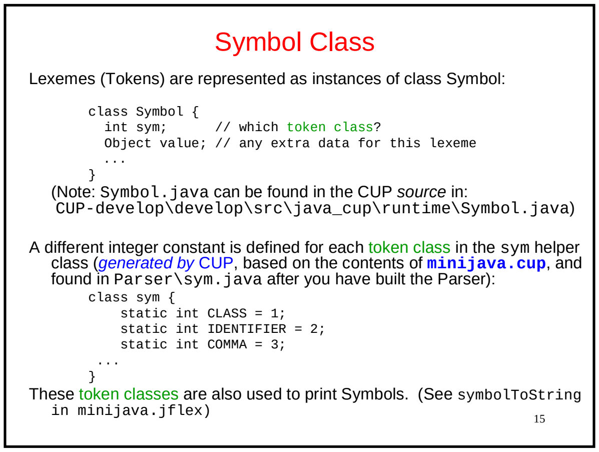
Automatic Scanner Generation in MiniJava

We use the `jflex` tool to automatically create a **scanner** from a specification file: `Scanner/minijava.jflex`

(We will use the CUP tool to automatically create a **parser** from a specification file: `Parser/minijava.cup`

CUP will also generate other code (e.g. the `sym` class) that we will use in the scanner, via the `Symbol` class.)

The MiniJava `Makefile` automatically rebuilds the scanner (or parser) whenever its specification file changes.



Symbol Class

Lexemes (Tokens) are represented as instances of class Symbol:

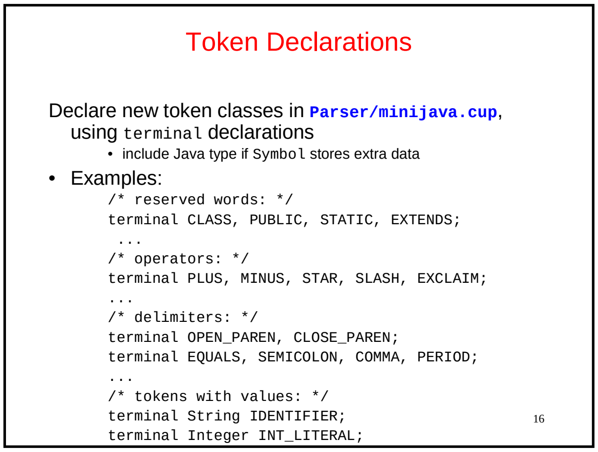
```
class Symbol {
  int sym;      // which token class?
  Object value; // any extra data for this lexeme
  ...
}
```

(Note: `Symbol.java` can be found in the CUP *source* in: `CUP-develop\develop\src\java_cup\runtime\Symbol.java`)

A different integer constant is defined for each token class in the `sym` helper class (*generated by* CUP, based on the contents of `minijava.cup`, and found in `Parser\sym.java` after you have built the Parser):

```
class sym {
    static int CLASS = 1;
    static int IDENTIFIER = 2;
    static int COMMA = 3;
 ...
}
```

These token classes are also used to print Symbols. (See `symbolToString in minijava.jflex`)



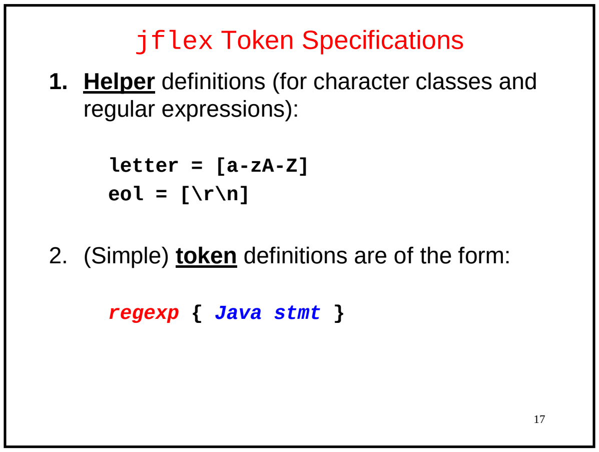
Token Declarations

Declare new token classes in `Parser/minijava.cup`, using `terminal` declarations

- include Java type if `Symbol` stores extra data

- Examples:

```
/* reserved words: */
terminal CLASS, PUBLIC, STATIC, EXTENDS;
 ...
/* operators: */
terminal PLUS, MINUS, STAR, SLASH, EXCLAIM;
...
/* delimiters: */
terminal OPEN_PAREN, CLOSE_PAREN;
terminal EQUALS, SEMICOLON, COMMA, PERIOD;
...
/* tokens with values: */
terminal String IDENTIFIER;
terminal Integer INT_LITERAL;
```



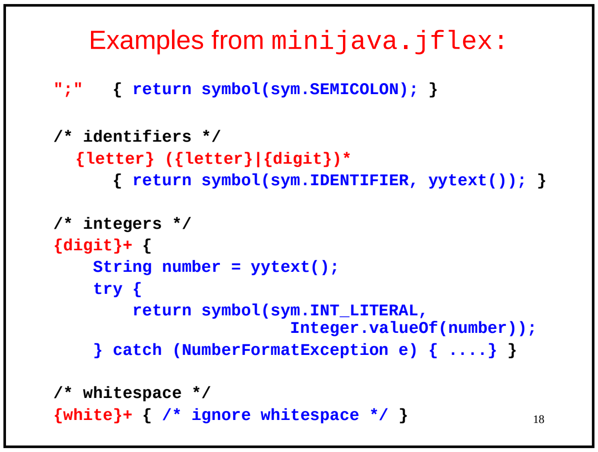
`jflex` Token Specifications

1. **Helper** definitions (for character classes and regular expressions):

```
letter = [a-zA-Z]
eol = [\r\n]
```

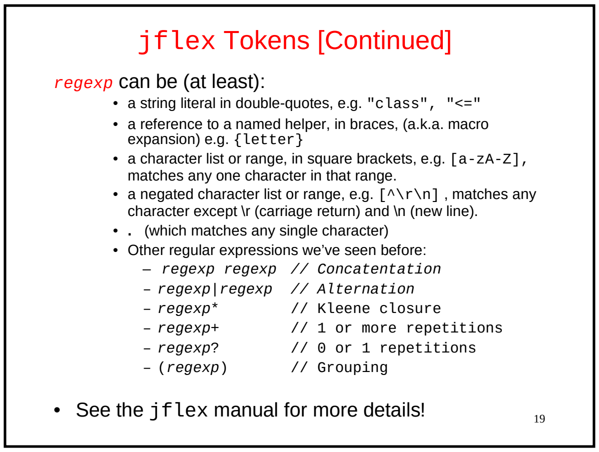
2. (Simple) **token** definitions are of the form:

regexp **{** ***Java stmt*** **}**



Examples from `minijava.jflex:`

```
";"   { return symbol(sym.SEMICOLON); }

/* identifiers */
  {letter} ({letter}|{digit})*
     { return symbol(sym.IDENTIFIER, yytext()); }

/* integers */
{digit}+ {
    String number = yytext();
    try {
        return symbol(sym.INT_LITERAL,
                      Integer.valueOf(number));
    } catch (NumberFormatException e) { ....} }

/* whitespace */
{white}+ { /* ignore whitespace */ }
```



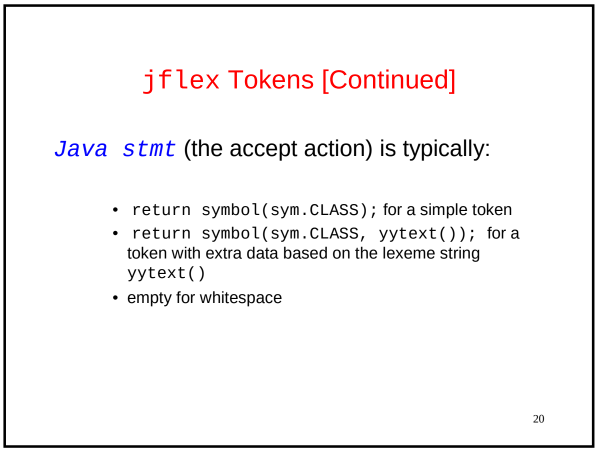
`jflex` Tokens [Continued]

`regexp` can be (at least):

- a string literal in double-quotes, e.g. `"class", "<="`
- a reference to a named helper, in braces, (a.k.a. macro expansion) e.g. `{letter}`
- a character list or range, in square brackets, e.g. `[a-zA-Z],` matches any one character in that range.
- a negated character list or range, e.g. `[^\r\n]` , matches any character except \r (carriage return) and \n (new line).
- `.` (which matches any single character)
- Other regular expressions we've seen before:

```
– regexp regexp  // Concatentation
– regexp|regexp  // Alternation
– regexp*        // Kleene closure
– regexp+        // 1 or more repetitions
– regexp?        // 0 or 1 repetitions
– (regexp)       // Grouping
```

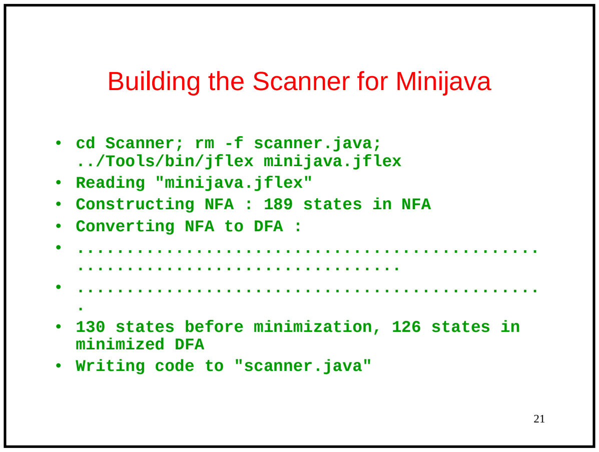
- See the `jflex` manual for more details!



`jflex` Tokens [Continued]

`Java stmt` (the accept action) is typically:

- `return symbol(sym.CLASS);` for a simple token
- `return symbol(sym.CLASS, yytext());` for a token with extra data based on the lexeme string `yytext()`
- empty for whitespace



Building the Scanner for Minijava

```
• cd Scanner; rm -f scanner.java;
  ../Tools/bin/jflex minijava.jflex
• Reading "minijava.jflex"
• Constructing NFA : 189 states in NFA
• Converting NFA to DFA :
• ...............................................
  ...................................
• ...............................................
  .
• 130 states before minimization, 126 states in
  minimized DFA
• Writing code to "scanner.java"
```