CSC 330 Lecture Notes Week 2
Intro to Programming Language Translation
Intro to JFlex

I. Overview of high-level I/O specs for a programming language translator

Major Module	In	Out
Lexical Analyzer	Source Code	Token Stream
Parser	Token Stream	Parse Tree Symbol Table
Code Generator	Parse Tree Symbol Table	Object Code

II. "Compiler" versus "Translator"

A. The view of a compiler as a monolith whose only job is to generate object code has largely faded in a world of integrated development environments (IDEs).

B. Frequently, programming language translators are expected to function as a collection of modular components in environments that provide a host of capabilities, only one of which is pure object code generation.

C. Consider the comparison diagram in Figure 1.

III. "Compiler" versus "Interpreter"

A. The traditional distinction between compiler and interpreter is that a compiler generates machine code to be run on hardware, whereas an interpreter executes a program directly without generating machine code.

B. However, this distinction is not always clear cut, since:

1. Many interpreters do some degree of compilation, producing code for some virtual rather than physical machine, e.g., Java's virtual machine (JVM).
2. Compilers for languages with dynamic features (such as Lisp, SmallTalk, and even C++) have large run-time support systems, amounting in some cases to essentially the same code found in interpreters.

C. Consider again the diagram in Figure 1.

IV. Rationale for non-monolithic translator design:

A. Machine Independence and Component Reuse

1. A basic *front end*, consisting of a lexer and parser, translates from input source string to some *machine-independent* internal form, such as a parse tree.
2. A number of machine-specific *back ends* (i.e., machine-specific code generators) can then be built to generate code from this internal form.
3. This design encapsulates a large, reusable portion of the translator into the machine-independent front end.
4. You'll look at how back ends are added to front ends if you take CSC 431.

B. Support for Both Interpretation and Compilation

1. A machine-independent internal form can also be fed into an interpreter, to provide immediate execution.
2. Machine independent forms are more amenable to *incremental* compilation, where local changes to a source program can be locally recompiled and linked, without requiring complete global recompilation.
3. We'll do interpretation in CSC 330, but not compilation to machine code.

C. Tool Support

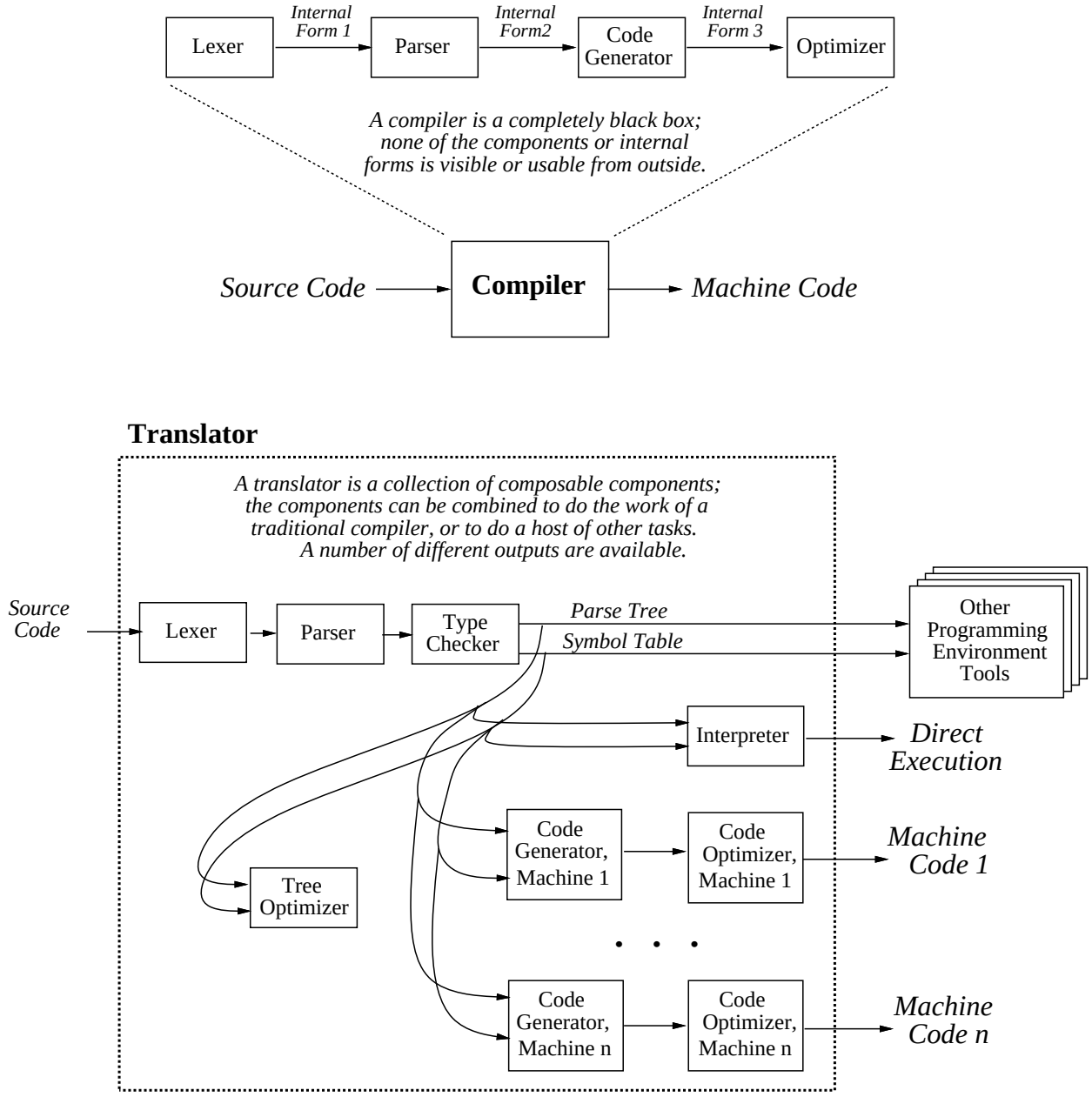


Figure 1: Compiler Versus Translator.

1. There are a host of tools that can deal better with a *syntax-oriented* internal form than with the *code-oriented* internal forms produced by monolithic compilers.
2. These tools include:
   a. pretty printers -- automatic program "beautifiers" that indent programs nicely, highlight keywords, etc.
   b. structure editors -- "smart" editors that automatically maintain the syntactic correctness of a program as it's being edited (e.g., "end" is automatically inserted by the editor whenever a "begin" is typed).
   c. program cross reference tools
   d. source-to-source program transformation tools, e.g, translate C into Java, or vice versa

e. program understanding tools, such as program animators (discussed in the Software Engineering class)

f. static and dynamic analysis tools, such as data-flow analyzers, complexity measures, and concurrency analyzers

V. A very high-level description of what the main components of a translator do.

A. Consider the following simple program, in a Pascal-subset programming language we will use in 330 examples:

```
program
    var X1, X2: integer;        { integer var declaration }
    var XR1: real;              { real var declaration }

begin
    XR1 := X1 + X2 * 10;        { assignment statement }
end.
```

B. Lexical analysis of the simple program

1. What the lexical analyzer does is scan the program looking for major lexical elements, called *tokens*
2. Tokens include constructs such as keywords, variable names, and operation symbols.
3. Tokens are program elements just above the level of raw characters.
4. The lexer takes the program source text as input, and produces a sequences of tokens.
5. In this example, lexical analysis would produce the following token sequence: `program, var, X1, ‘,’, X2, ‘:’, integer, ‘;’, var, XR2, ‘:’, real, ‘;’, begin, XR1, ‘:=’, X1, ‘+’, X2 ‘*’, 10 ‘;’, end, ‘.’`
6. Note that the lexer discards *whitespace* -- blanks, tabs, newlines, and comments.
7. You will write a lexical analyzer in Assignment 2.

C. Parsing the simple program

1. The next step in the translation is to analyze the grammatical structure of the program, producing a *parse tree*.
2. The grammatical analysis basically consists of checking that the program is syntactically legal
   a. Such legality is based on the formal grammar of the language in which the program is written.
   b. Formal grammars are typically expressed in BNF or some equivalent notation, as we discussed last week.
   c. The parser reads tokens one at a time from the lexer, and checks that each token "fits" in the grammar.
      i. This "fitting" (i.e., parsing) is the most complicated part of the whole translation.
      ii. You will write a parser in Assignment 3.
3. The resulting parse tree records the grammatical structure of the program in a form that is convenient for use by the next phase of the translation.
4. The parse tree for the simple example program above looks something like the picture in Figure 2.

D. The final major translation step of a compiler is code generation

1. The code generator performs a standard tree traversal (end-order typically) to produce machine code for the program.
2. Suppose we have a stack-oriented machine with instructions like PUSH, ADD, MULT, and STORE.
3. The code generated from the above parse tree would look something like this:

```
PUSH X2
PUSH 10
MULT
PUSH X1
ADD
PUSH @XR1
STORE
```

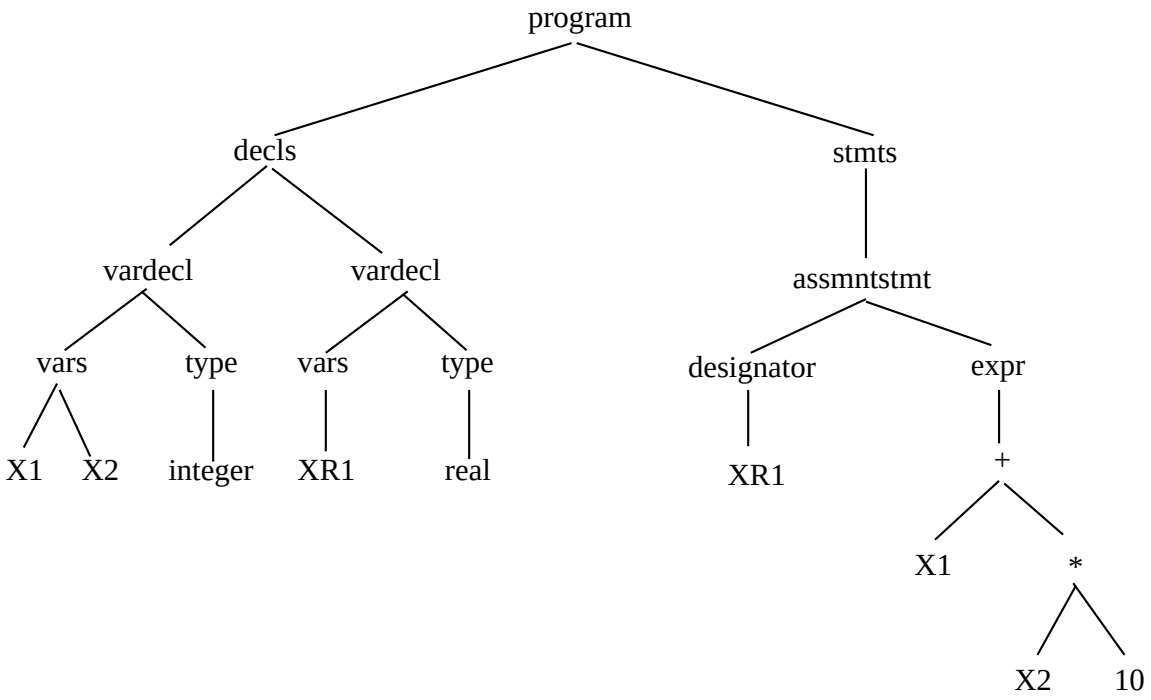


Figure 2: Parse Tree for Sample Program.

4. This type of code is generated by translating the tree operations into machine operations as the end-order tree traversal commences.
5. Instead of code generation, you will write tree interpreter in 330 Assignment 4.
6. You will also write a functional form of interpreter in Assignment 7.

E. A major data component used by all three translation steps is a *symbol table*

1. The symbol table is created by the lexer and parser as identifiers are recognized; it is used by the code generator.
2. The table records identifier information, such as types, as well as the fact that identifiers are properly declared.
3. This information is used to perform type checking and other checks, such as that identifiers are declared before they are used, etc.
4. A symbol table for the above simple example would look like this:

Symbol	**Class**	**Type**
X1	var	integer
X2	var	integer
XR1	var	real

VI. A first look at *Meta*-Translation

A. The first meta-translation tool we'll use in 330 is JFlex (Java Fast LEXical analyzer generator); it is the latest in a series of tools based on Lex, the original lexical analyzer generator.

B. The other meta-translation tool we'll be using is named CUP (Construction of Useful Parsers); it is the latest in a series of tools based on YACC (Yet Another Compiler-Compiler).

C. What a meta-translator does is accept a high-level description of a program, and then generate the program that fits the description.

D. That is, a meta-translator is a program that produces another program as its output.
   1. A meta-translator does not do the translation itself, but rather generates a program that does it.
   2. E.g., JFlex does not do lexical analysis -- it generates a Java program that does the lexical analysis.
   3. CUP does not do parsing -- it generates a Java program that does it.
   4. This concept will probably take a while to wrap your brain around.

E. The following table illustrates some common meta-translators, including JFlex and CUP that we will be using:

INPUT	TOOL	OUTPUT
Regular Expressions	Lexical Analyzer Generator (E.g, JFlex,LEX)	*Lexical Analyzer*
(Restricted) BNF Grammar *"Canned" Parsing Engine*	Parser Generator (E.g., CUP,YACC)	*Language-Specific Parser*
Machine Description Grammar	Code Generator Generator (E.g., CODEGEN)	*Machine-Specific Code Generator*
Reg Exprs + *BNF* + *Semantic Actions* + *Engine*	Generic Compiler-Compiler (E.g, YACC+LEX, PQCC, others)	*Language-Specific Compiler*
All of the above inputs + *canned tool templates*	Environment Generator (E.g, PECAN)	*Language-Specific Programming Environment*

VII. A simple programming language for the 330 examples

A. In this and forthcoming lectures we will be studying how Lex and Yacc work by focusing on a very simple programming language.

B. Here is an EBNF grammar for the subset of Pascal we will use in 330 examples:

```
program        ::= PROGRAM block '.'
block          ::= decls BEGIN stmts END
decls          ::= [ decl { ';' decl } ]
decl           ::= typedecl | vardecl | procdecl
typedecl       ::= TYPE identifier '=' type
type           ::= identifier | ARRAY '[' integer ']' OF type
```

```
vardecl        ::= VAR vars ':' type
vars           ::= identifier { ',' identifier }
procdecl       ::= prochdr ';' block
prochdr        ::= PROCEDURE identifier '(' formals ')' [ ':' identtype ]
formals        ::= [ formal { ';' formal } ]
formal         ::= identifier ':' identifier
stmts          ::= stmt { ';' stmt }
stmt           ::=  | assmntstmt | ifstmt | proccallstmt | compoundstmt
assmntstmt     ::= designator ':=' expr
ifstmt         ::= IF expr THEN stmt [ ELSE stmt ]
proccallstmt   ::= identifier '(' exprlist ')'
compoundstmt   ::= BEGIN stmts END
expr           ::= integer | real | char | designator |
                   identifier '(' exprlist ')' | expr relop expr |
                   expr addop expr | expr multop expr | unyop expr |
                   '(' expr ')'
designator     ::= identifier { '[' expr ']' }
exprlist       ::= [ expr { ',' expr } ]
relop          ::= '<' | '>' | '=' | '<=' | '>=' | '<>'
addop          ::= '+' | '-' | OR
multop         ::= '*' | '/' | AND
unyop          ::= '+' | '-' | NOT
```

VIII. Overview of a JFlex-generated lexical analyzer

A. Recall from above that a lexical analyzer takes in a source program and produces a stream of *tokens*, consisting of program keywords, identifiers, operator symbols, and similar "word-sized" items in the program.

B. Writing a program to do such lexical analysis can be a rather tedious process that requires scanning the program a character at a time, constructing tokens as certain character patterns are recognized.

1. For example, when the lexical analyzer sees an alphabetic character, it scans ahead looking for additional alphanumeric characters, until it finds something other than an alphanumeric character.
2. This particular scanning process recognizes an identifier -- a pattern of characters starting with a letter followed by zero or more letters and/or digits.
3. As another example, when a Pascal lexical analyzer sees a ':', it scans ahead to see if the next character is a '=', and if so returns a ":=" token, otherwise it returns a ":" token and continues the scan after the ':'.

C. When the analyzer recognizes a known token, it outputs a numeric value that corresponds to the token.

1. The numeric token values are just some consistent numeric definitions for all the kinds of tokens that can appear in a program.
2. For example, here are some of the JFlex token definitions for the simple language defined above:

```
public static final int DIVIDE = 18;
public static final int CHAR = 37;
public static final int SEMI = 19;
public static final int INT = 35;
public static final int ARRAY = 3;
public static final int LESS = 27;
public static final int MINUS = 17;
 . . .
```

3. We'll talk in a bit about why these particular names and values are used by JFlex, but the point is that any set of token values will do as long as they're used consistently. (Question: why are the numbers seemingly randomly assigned?)

D. The form of scanning described just above involves two essential forms of analysis:

1. The recognition of certain patterns in the input (such as identifiers, operators, etc.)
2. A set of rules, or a table of some sort, that indicates what token value to produce when a particular pattern is recognized.

E. What JFlex does is to allow a programmer to specify these two essential forms of scanning in a high-level

form, rather than in low-level Java code.

F. That is, a JFlex specification consists of two essential components:
   1. A high-level description of the various patterns that describe what tokens to recognize
   2. A set of rules that specify precisely what token values to produce

IX. A concrete Lex example

A. We will proceed at this point to a concrete example of what a Lex specification looks like for the simple programming language defined above.

B. After we look at the example, we'll consider a bit what's going on behind the scenes with Lex, and the formal underpinnings of lexical analyzers in general.

C. Attached to the notes are listings for the following files:

- `pascal.jflex` -- the JFlex lexer specification
- `PascalLexerTest.java` -- a simple testing program
- `sym.java` -- token definitions
- `pascal-tokens.cup` -- CUP file used to generate token definitions

X. Highlights of the example

A. The general format of a JFlex specification is

auxiliary code -- typically not used except for imports
%%
pattern definitions -- including internally-used methods in `%{ ... %}`
%%
lexical rules

B. Patterns are specified in a simple regular expression language, with the following operators:

x	the character "x"
"x"	an "x", even if x is an operator.
\x	an "x", even if x is an operator.
[xy]	the character x or y.
[x−z]	the characters x, y or z.
[^x]	any character but x.
.	any character but newline.
^x	an x at the beginning of a line.
<y>x	an x when Lex is in start condition y.
x$	an x at the end of a line.
x?	an optional x.
x*	0,1,2, ... instances of x.
x+	1,2,3, ... instances of x.
x|y	an x or a y.
(x)	an x.
x/y	an x but only if followed by y *(CAREFUL)*.
{xx}	the translation of xx from the definitions section.
x{m,n}	*m* through *n* occurrences of x

C. A JFlex rule is of the form:

pattern *action*

where a *pattern* is in the regular expression language and an *action* is a Java statement, most typically a compound statement in "`{`", "`}`" braces.

D. The function *next_token*

1. It's the name chosen by JFlex, so we have to live with it.
2. Its return value is the representation of a token, which in our case is defined by the CUP-supplied class `Symbol.java`; it contains four public data fields of interest to a lexer:

```
/** The symbol number of the token being represented */
public int sym;

/** The left and right character positions in the source file
    (or alternatively, the line and column number). */
public int left, right;

/** The auxiliary value of a token such as an identifier string,
    or numeric token value. */
public Object value;
```

3. `next_token` reads its input from from a character stream supplied from outside, such as a file or standard-input stream.

E. The String-valued method `yytext()`.

1. The string value of a recognized token is returned by `yytext`.
2. It's used when the analyzer needs to return information in addition to the numeric token value, e.g., the text of an identifier or value of an integer.

XI. Running JFlex

A. The most typical form of JFlex invocation is command-line.

B. It takes a **.jflex* file as input and produces the file `Yylex.java` by default.

1. You can also supply the JFlex "`%class`" directive to specify the name of the generated `.java` file, and therefore, the name of the lexer class.
2. E.g., "`%class PascalLexer`" is used in the Pascal lexer example.

C. To build the executable lexical analyzer, compile `Yylex.java` (or whatever you named it with the `%class` directive) with the Java compiler.

D. For example, here is how to compile the stand-alone Pascal lexer, its driver, and the necessary symbol-definition file:

```
jflex pascal.jflex
javac PascalLexer.java PascalLexerTest.java sym.java
```

E. Typically the compilation of Lex and Java files is done using a *make* with a *Makefile*, or an equivalent Windows batch make file.

1. The 330/examples/jflex directory includes a Makefile for the Pascal lexer.
2. To run Lex and compile all the pieces of the stand-alone example, simply type "make" at the UNIX shell; this will run all the necessary steps; these steps are:

```
cup pascal-tokens.cup
jflex pascal.jflex
javac PascalLexer.java PascalLexerTest.java sym.java
java PascalLexerTest $*
```

3. We will talk about the details of this in an in-class demonstration.

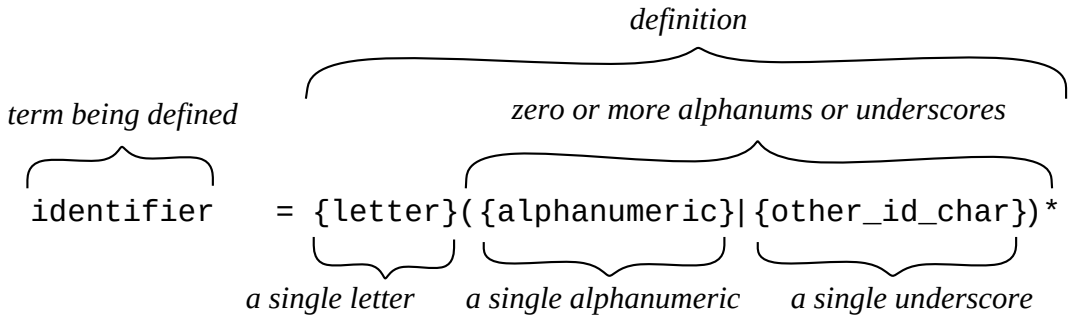
XII. JFlex documentation, examples, and download.

A. The Assignment 2 writeup describes where to get JFLex documentation and executable program.

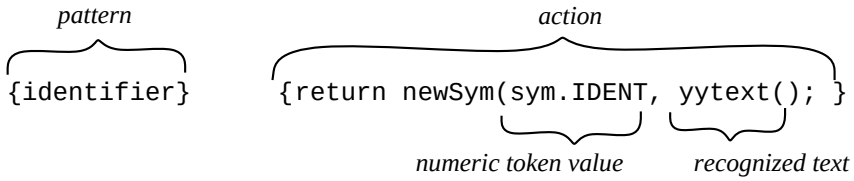
B. The 330 `doc` directory has links to a number of additional documentation sources, with useful examples.

C. Additional JFlex examples can be found in 330/examples/jflex.

XIII. A closer look `pascal.jflex` patterns, identifier in particular, on line 55.



XIV. A closer look at `pascal.jflex` rules, identifier in particular, on lines 103.



XV. A closer look at `sym.java`

A. As the comment at the top indicates, this was generated by CUP.

B. The input to CUP is the file `pascal-tokens.cup`.

C. For Assignment 2, you can write a comparable `.cup` file for your EJay lexer and generate `sym.java` using CUP.

D. Alternatively, you can hand build a `sym.java` file, using the Pascal version as a pattern.

E. What is critical is that numbers are unique for each token, and that number 0 is not used, since it's the value used to indicate end of input.

XVI. A closer look at PascalLexerTest.java

A. It's a simple loop that calls `next_token` and prints out what is returned.

B. Note that input is from a command-line filename.

C. Sample Pascal input files are in `330/examples/*.p`.

XVII. On case sensitivity

A. In order to recognize different character cases, they must be explicitly accounted for in the lexer.

B. E.g., to recognize both upper and lowercase keywords in in some language (not EJay), one would uses rules of the form:

```
begin   { return newSym(sym.BEGIN); }
BEGIN   { return newSym(sym.BEGIN); }
```

C. To recognize any case combination in keywords, e.g., "bEGiN", one might employ a case-shifting preprocessor, or a case-shift-keyword-table-lookup function in the lex rule for {identifier}.

XVIII. The longest-match-preferred rule of JFlex.

A. As the JFlex manual describes, the rules for ambiguity resolution are:

• The longest match is preferred.

• Among rules which match the same number of characters, the rule given first is preferred.

1. E.g., suppose we have rules of the form

integer {/* action for keyword **integer** */}
{identifier} {/* action for identifiers, including *integer* */}

2. In this case, the keyword **integer** and the identifier *integer* are ambiguous.
3. The rules say that in cases such as that the input string "integer" will be recognized as a keyword, since that rule appears first.
4. The longest-match-preferred rule makes patterns with *'s at the end dangerous, since they will probably read farther ahead than you'd like (see Section 3 of the original Lex manual for more details).

XIX. On potential ambiguities

A. Suppose, as in Pascal, that we have both ":" and ":=" as tokens -- how are they recognized?

B. Consider the following two possibilities:

1. Possibility 1:

```
begin                { ... }
...
":="                 { return newSym(sym.ASSMNT); }
":"                  { return newSym(sym.COLON); }
...
```

2. Versus Possibility 2:

```
begin                { ... }
...
":"                  { return newSym(sym.COLON); }
":="                 { return newSym(sym.ASSMNT); }
...
```

3. Which one should be used, or is either OK? Why? (Think about it.)

XX. Attaching source locations to tokens

A. In any good compiler, when errors occur during compilation, the compiler associates a line and sometimes column number with each error message.

B. How does the compiler keep track of line/column numbers once the source code has been lexically analyzed?

C. The basic idea is that the lexer records this information and passes it onto the parser, where it is "hung off" of the parse tree for subsequent use.

D. In a JFlex lexer, these values are included in the `Symbol` value of a token.

E. Look at the implementation of the two `newSym` methods in `pascal.jflex`.

F. Unless source position information is recorded by the lexer and sent on through, it will be lost.

XXI. Whither comments?

A. For Assignment 2, you recognize comments, and print them out.

B. For Assignment 3, you will discard them altogether.

C. We'll discuss further in upcoming lectures.