Class 5

Lex Spec Example

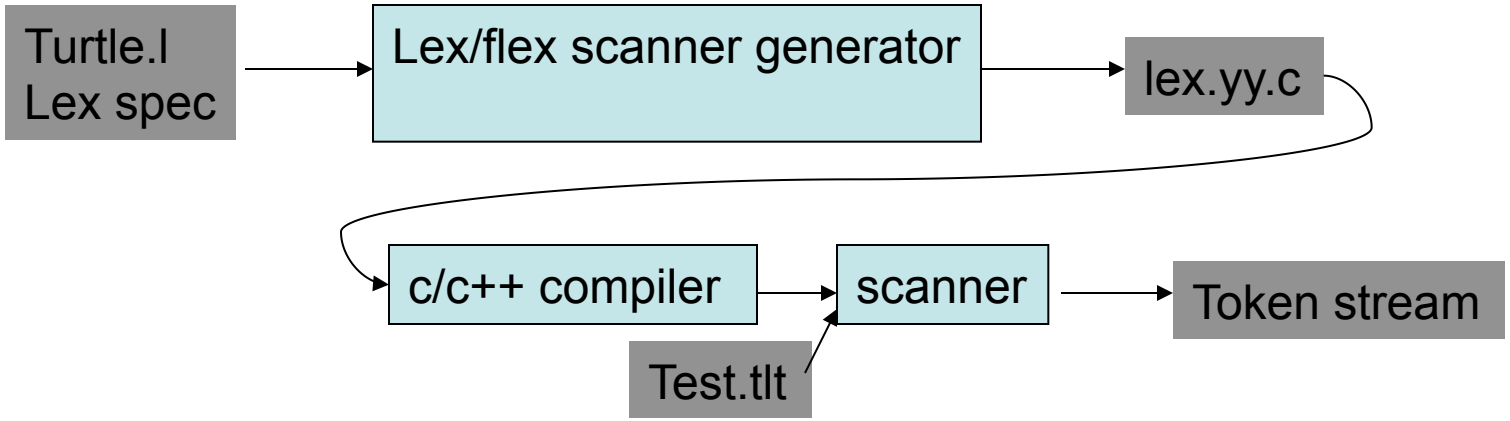
```
delim           [ \t\n]
ws              {delim}+
letter          [A-Aa-z]
digit           [0-9]
id              {letter}({letter}|{digit})*
number          {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
%%
{ws}            {/*no action and no return*?}
if              {return(IF);}
then            {return(THEN);}
{id}            {yylval=(int) installID(); return(ID);}
{number}        {yylval=(int) installNum(); return(NUMBER);}
%%

Int installID() {/* code to put id lexeme into string table*/}

Int installNum() {/* code to put number constants into constant table*/}
```

Some Notes on Lex

- yylval – global integer variable to pass additional information about the lexeme
- yyline – line number in input file
- yytext – returns the lexeme matched



Form of a JLex Spec File

user code

%%

JLex directives

%%

regular expression rules in the form of:

reg expr {action}

reg expr {action}

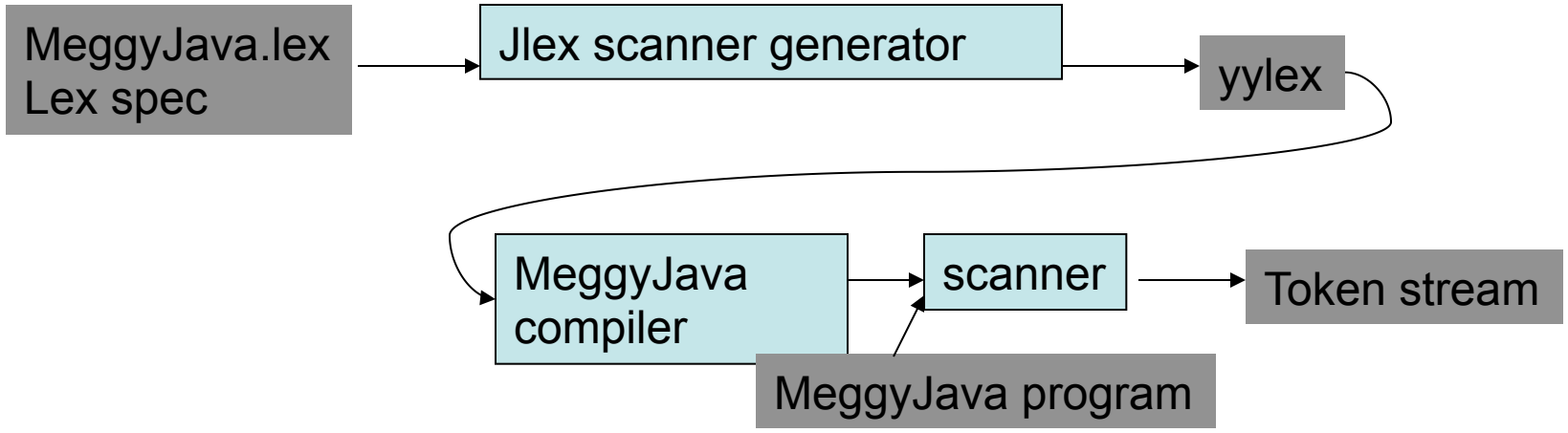
…

JLex Spec Example

```
class Token {
String text;
Token(String t){text = t;}
}
%%
Digit=[0-9]
AnyLet=[A-Za-z]
Others=[0-9'&.]
WhiteSp=[\040\n]
// Tell JLex to have yylex() return a Token
%type Token
// Tell JLex what to return when eof of file is hit
%eofval{
return new Token(null);
%eofval}
%%
[Pp]{AnyLet}{AnyLet}{AnyLet}[Tt]{WhiteSp}+  {return new Token(yytext());}
({AnyLet}|{Others})+{WhiteSp}+    {/*skip*/}
```

Some Notes on JLex

- yychar – character count matched
- yyline – line number in input file where matched
- yytext – returns the lexeme matched



A Java driver program that uses the scanner is:

```
import java.io.*;
class Main {
public static void main(String args[])
            throws java.io.IOException {
  Yylex lex = new Yylex(System.in);
  Token token = lex.yylex();
  while ( token.text != null ) {
     System.out.print("\t"+token.text);
     token = lex.yylex(); //get next token
   }
}}
```

Handling Ambiguities

What if

- $x_1 \ldots x_i \in L(R)$ and also
- $x_1 \ldots x_K \in L(R)$

Some examples?

Which token is used? How designated?

More Ambiguities

- What if
- • x1…xi ∈ L(Rj) and also
- • x1…xi ∈ L(Rk) ?

- Which token is used?

Lexical Error Detection and Handling

No rule matches a prefix of input ?

Problem: Compiler can’t just get stuck …

You should…
Do Some More Practice with reading and writing lex specs

How does the Scanner work under the Hood?

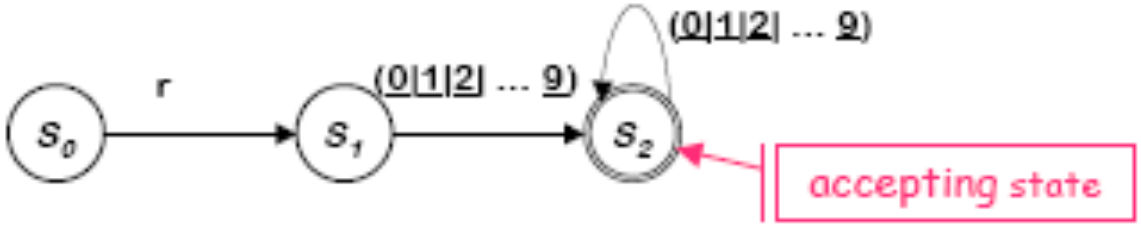
From Specification to Scanning…

Consider the problem of recognizing ILOC register names

Register → r (0|1|2| … | 9) (0|1|2| … | 9)*

- Allows registers of arbitrary number
- Requires at least one digit

RE corresponds to a recognizer (or DFA)



Recognizer for *Register*

Transitions on other inputs go to an error state, s_e

What is a Finite Automata?

Regular expressions = specification

Finite automata = implementation

A finite automaton consists of

- An input alphabet Σ
- A set of states S
- A start state n
- A set of accepting states $F \subseteq S$
- A set of transitions state $\rightarrow$input state

From Reg Expr to NFA

How do we build an NFA for:

a?

Concatenation? ab

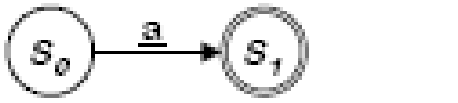
Alternation? a | b

Closure? a*

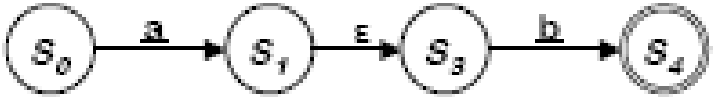
RE →NFA using Thompson's Construction

Key idea

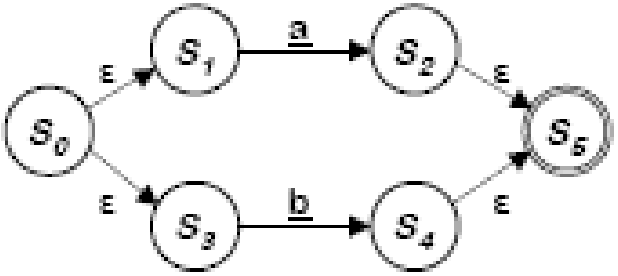
- NFA pattern for each symbol & each operator
- Join them with ε moves in precedence order



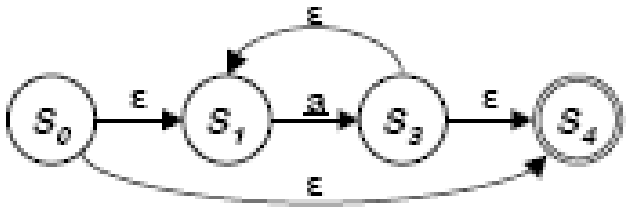
NFA for a



NFA for ab



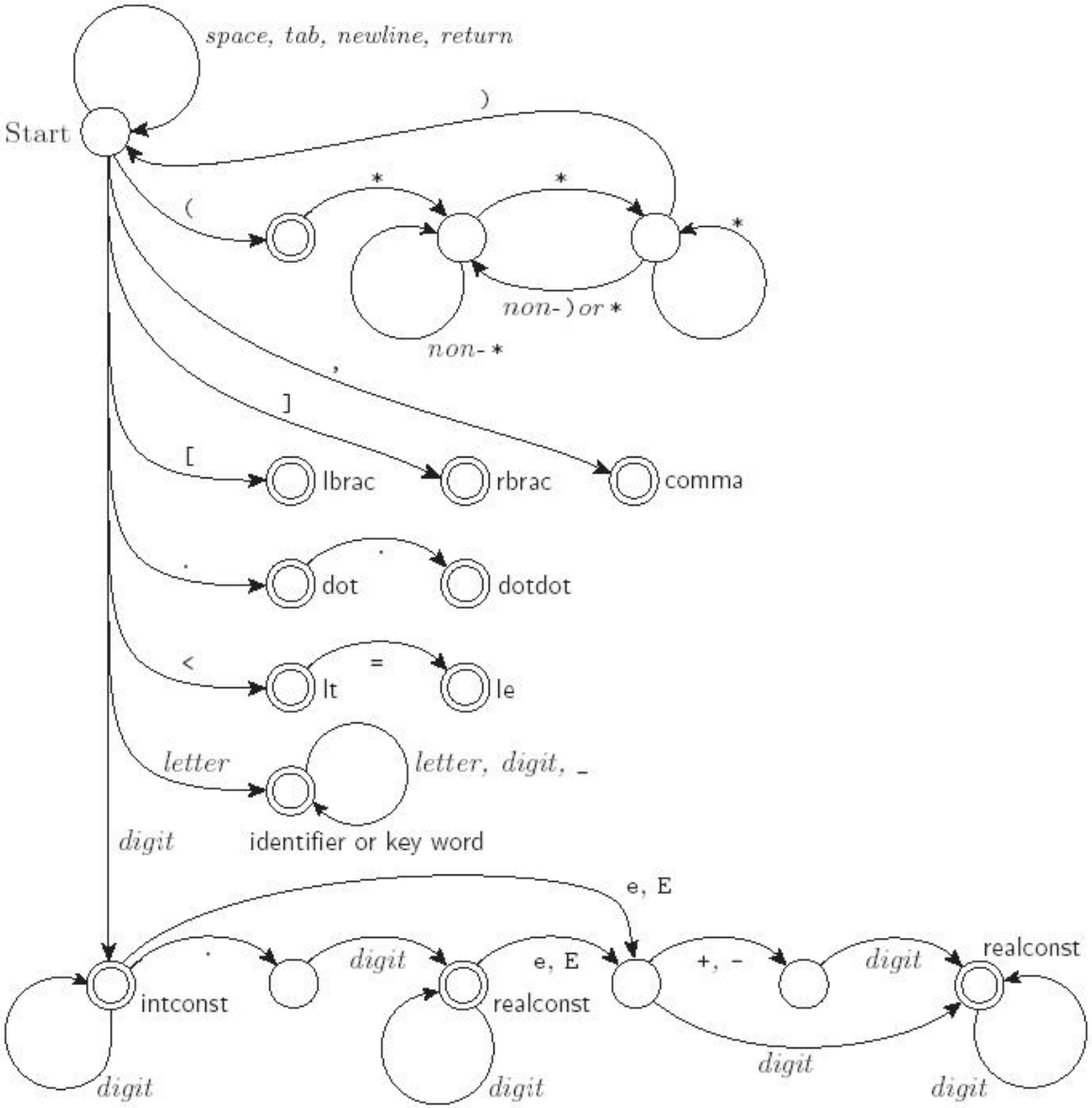
NFA for a | b



NFA for a*

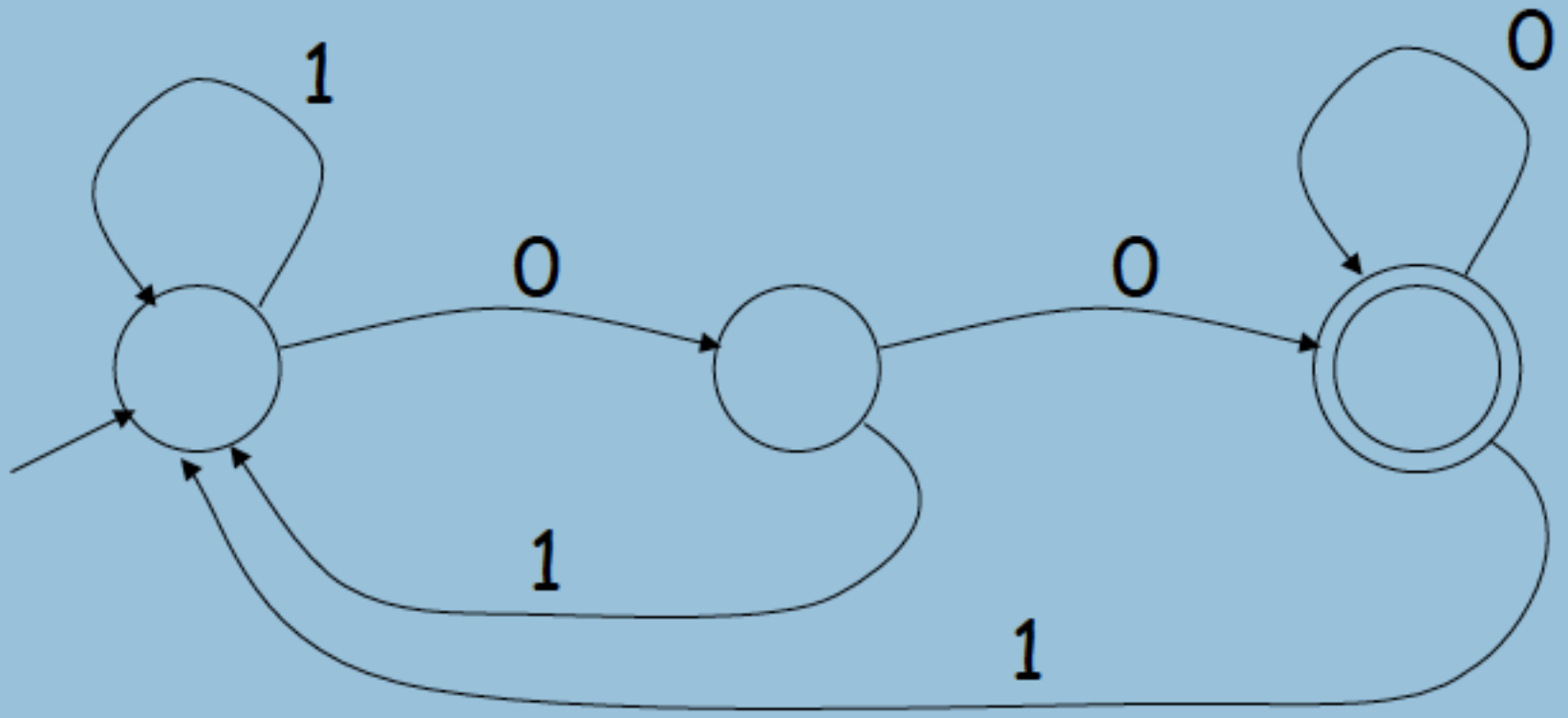
Ken Thompson, *CACM*, 1968

Scanning as a Finite Automaton



Understanding FA

- Alphabet {0,1}
- What language does this recognize?



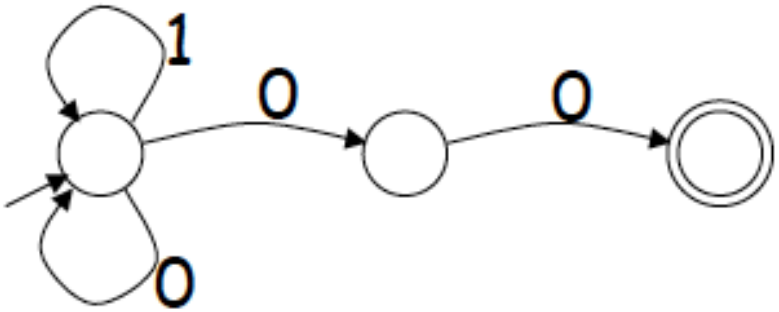
DFA vs NFA ?

- What is allowed?
- Which can be much bigger in size? Which is simpler?
- Which is faster to run?

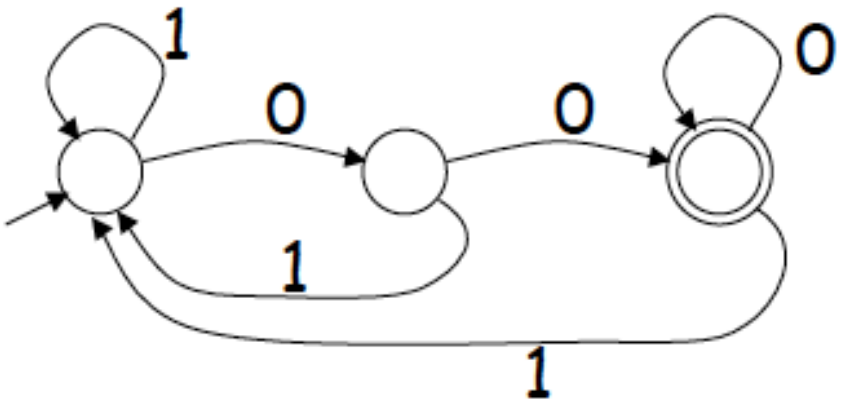
Comparison by size

For a given language NFA can be simpler than DFA

NFA



DFA



DFA can be exponentially larger than NFA

Automating Scanner Construction

To convert a specification into code:

1. Write down the RE for the input language
2. Build a big NFA
3. Build the DFA that simulates the NFA
4. Systematically shrink the DFA
5. Turn it into code

Scanner generators

- Lex and Flex work along these lines
- Algorithms are well-known and well-understood
- Key issue is interface to parser *(define all parts of speech)*
- You could build one in a weekend!

Implementing a DFA

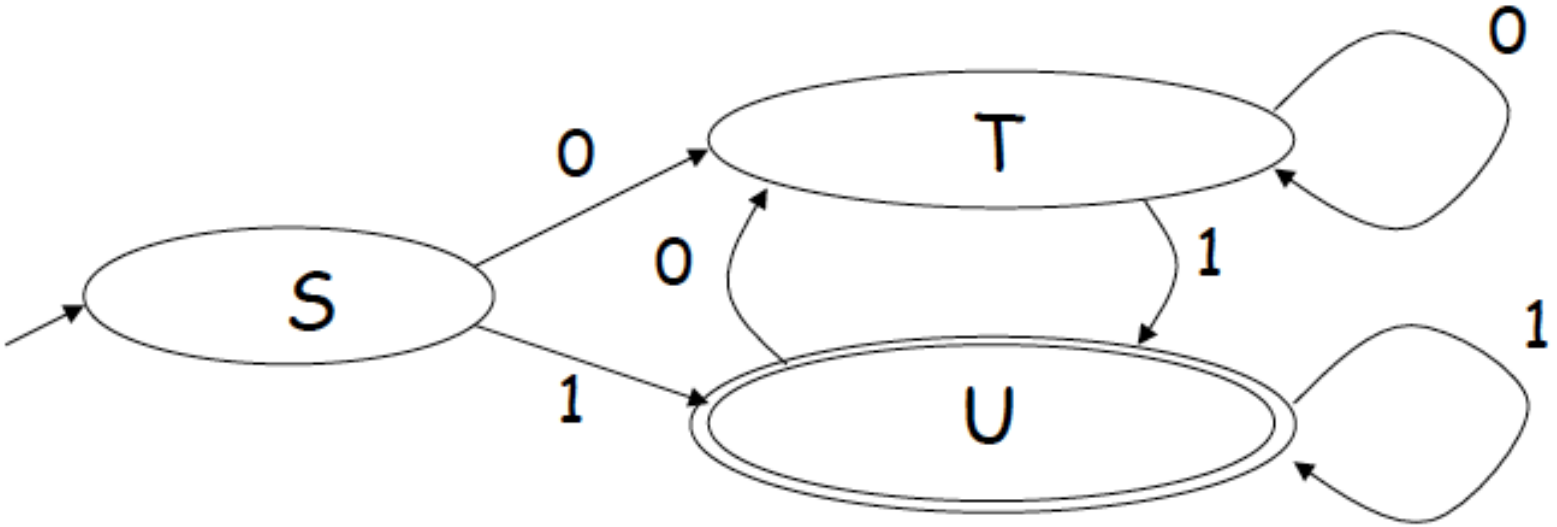
A DFA can be implemented by a 2D table T

- One dimension is "states"
- Other dimension is "input symbol"
- For every transition $S_i \rightarrow_a S_k$ define T[i,a] = k

DFA "execution"

- If in state S_i and input a, read T[i,a] = k and skip to state S_k
- Very efficient

Table Implementation of a DFA



	0	1
S	T	U
T	T	U
U	T	U

However, 3 Major Ways to Build Scanners

- ad-hoc
- semi-mechanical pure DFA
  (usually realized as nested case statements)
- table-driven DFA

- Ad-hoc generally yields the fastest, most compact code by doing lots of special-purpose things, though good automatically-generated scanners come very close

Manually written scanner code

```
current = START_STATE;
token = "";
// assume next character has been preloaded into a buffer
while (current != EX)
{
        int charClass = inputstream->thisClass();
        switch (current->action(charClass))
        {
                case SKIP:
                        inputstream->advance();break;
                case ADD:
                        char* t = token; int n = ::strlen(t);
                        token = new char[n + 2]; ::strcpy(token, t);
                        token[n] = inputstream->thisChar(); token[n+1] = 0;
                        delete [] t; inputstream->advance(); break;
                case NAME:
                        Entry * e = symTable->lookup(token);
                        tokenType = (e->type==NULL_TYPE ? NAME_TYPE : e->type);
                        break;
                ...
        }
        current = current->nextState(charClass);
}
```

In summary, Scanner is the only phase to see the input file, so…

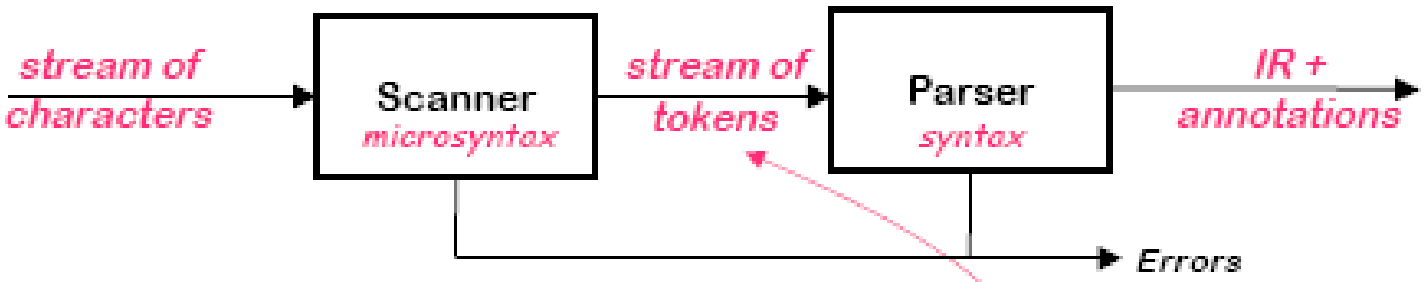
The scanner is responsible for what?

In summary, Scanner is the only phase to see the input file, so…

The scanner is responsible for:

- tokenizing source
- removing comments
- saving text of identifiers, numbers, strings
- saving source locations (file, line, column) for error messages

Why separate phases?



Scanner is only pass that touches every character of the input.

Why separate the scanner and the parser?

- Scanner classifies words
- Parser constructs grammatical derivations
- Parsing is harder and slower
- Separation simplifies implementation
  - smaller grammar for parser
  - faster front end

token is a pair
<part of speech, lexeme>

More Details on Lex/Flex (for your own reading pleasure)

A Makefile for the scanner

```
eins.out: eins.tlt scanner
	scanner < eins.tlt > eins.out

lex.yy.o:  lex.yy.c token.h symtab.h
	gcc -c lex.yy.c

lex.yy.c: turtle.l
	flex turtle.l

scanner:  lex.yy.o symtab.c
	gcc lex.yy.o symtab.c -lfl -o scanner
```

A typical token.h file

```
#define SEMICOLON 274
#define PLUS 275
#define MINUS 276
#define TIMES 277
#define DIV 278
#define OPEN 279
#define CLOSE 280
#define ASSIGN 281
…  /*for all tokens*/

typedef union YYSTYPE
{ int i; node *n; double d;}
      YYSTYPE;
YYSTYPE yylval;
```

A typical driver for testing the scanner without a parser

```
%%

main(){
int token;

while ((token = yylex()) != 0) {

switch (token) {
     case JUMP : printf("JUMP\n"); break;
/*need a case here for every token possible, printing yylval as needed for
     those with more than one lexeme per token*/
      default:
            printf("ILLEGAL CHARACTER\n"); break;



}
}
}
```