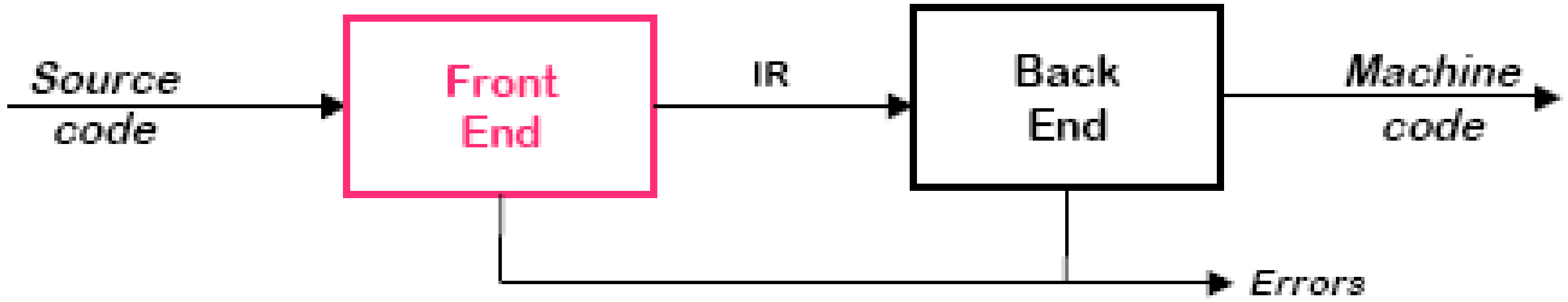
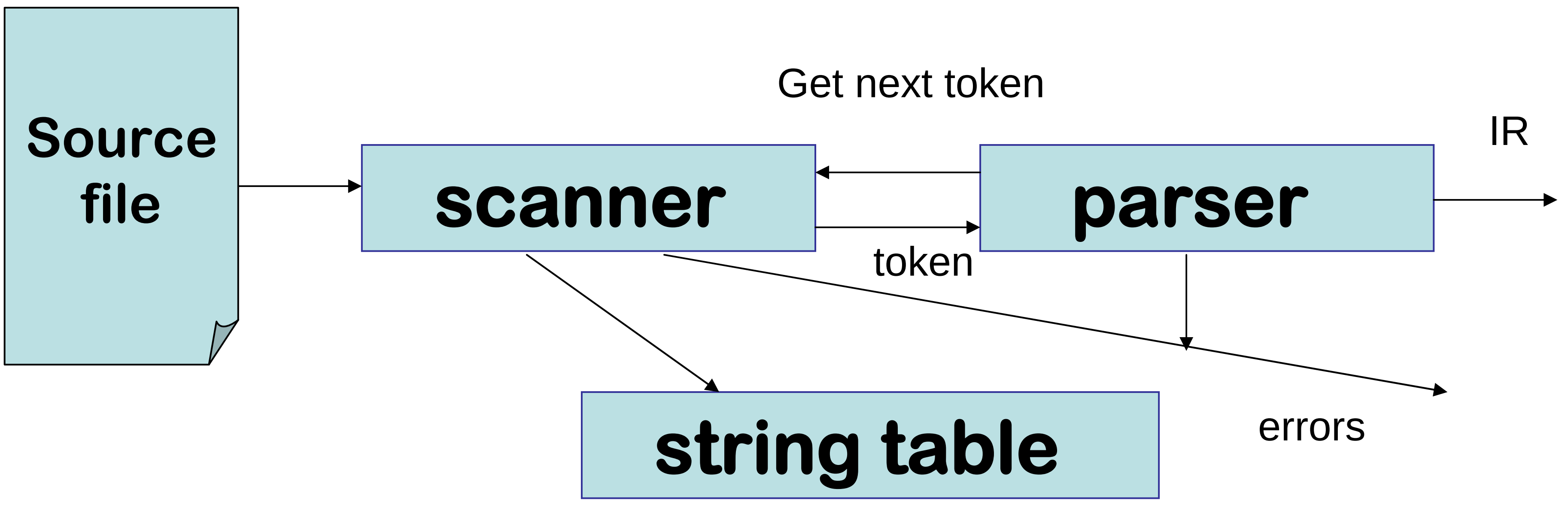
The Front End: Scanning and Parsing



How they work together…



What is a token? A lexeme?

- English?
- Programming Languages?

- Lexeme
- Token
- Examples?

lexemes	tokens

Designing a Scanner

Step 1: define a finite set of tokens

How?

Step 2: describe the strings (lexemes) for each token

How?

So, a simple scanner design?

Then, why did they invent lex?

Poor language design can complicate scanning

- Reserved words are important

  if then then then = else; else else = then (PL/I)

- Insignificant blanks (Fortran & Algol68)

  do 10 i = 1,25

  do 10 i = 1.25

- String constants with special characters (C, C++, Java, ...)

  newline, tab, quote, comment delimiters, ...

- Finite closures (Fortran 66 & Basic)
  - Limited identifier length
  - Adds states to count length

Even, simple examples: i vs if ; = vs ==

It is not so straightforward…

Specifying lexemes with Regular Expressions

Let Σ be an alphabet.
Rules for Defining regular expressions over Σ :

- ε Denotes the set containing the empty string.
- For each a in Σ , a is the reg expr denoting {a}

- If r and s are reg expr's, then

r s	= set of strings consisting of strings from r followed by strings from s
r | s	= set of strings for either r or s
r *	= 0 or more strings from r (closure)
(r)	used to indicate precedence

Examples of Regular Expressions for Lexemes

Identifiers:

Letter → (a|b|c| … |z|A|B|C| … |Z)

Digit → (0|1|2| … |9)

Identifier → *Letter* (*Letter* | *Digit*)*

shorthand for

(a|b|c| … |z|A|B|C| … |Z) ((a|b|c| … |z|A|B|C| … |Z) | (0|1|2| … |9))*

Numbers:

Integer → (+|-|ε) (0| (1|2|3| … |9)(*Digit* *))

Decimal → *Integer* . *Digit* *

Real → (*Integer* | *Decimal*) E (+|-|ε) *Digit* *

Complex → (*Real* , *Real*)

Numbers can get much more complicated!

Using symbolic names does not imply recursion

underlining indicates a letter in the input stream

What strings/lexemes are represented by these regular expressions?

Practice with writing regular expressions

- Binary numbers of at least one digit
- Capitalized words
-Legal identifiers that must start with a letter, can contain either upper or lower case letters, digits, or _.
-white space including tabs, newlines, spaces

Shorthand for regular expressions?

What strings are accepted here?

- Numerical literals in Pascal may be generated by the following:

$$
\begin{aligned}
digit &\longrightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
unsigned\_integer &\longrightarrow digit\ digit\ * \\
unsigned\_number &\longrightarrow unsigned\_integer\ ((\ \texttt{.}\ \ unsigned\_integer) \mid \epsilon) \\
&\quad (((\texttt{e} \mid \texttt{E})(\texttt{+} \mid \texttt{-} \mid \epsilon)\ unsigned\_integer) \mid \epsilon)
\end{aligned}
$$

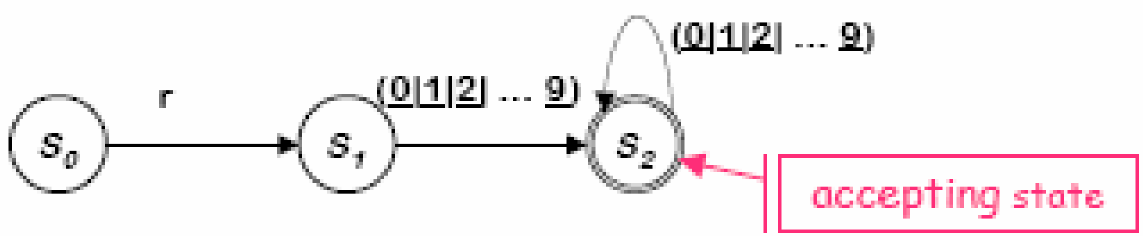
From Specification to Scanning…

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

From Reg Expr to NFA

How do we build an NFA for:

a?

Concatenation? ab

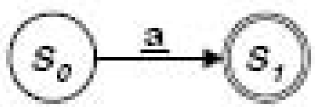
Alternation? a | b

Closure? a*

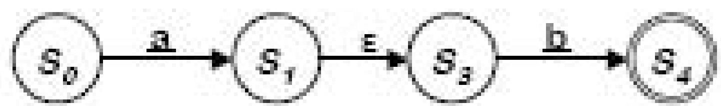
RE →NFA using Thompson's Construction

Key idea

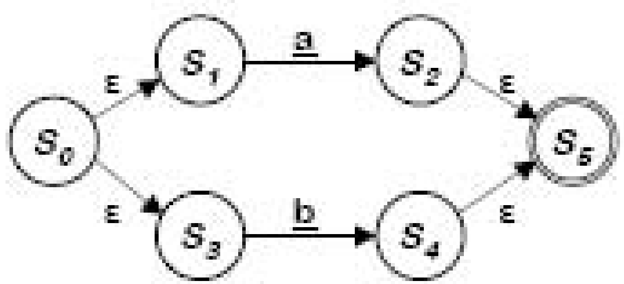
- NFA pattern for each symbol & each operator
- Join them with ε moves in precedence order



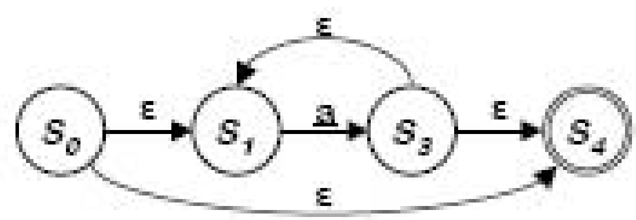
NFA for a



NFA for ab



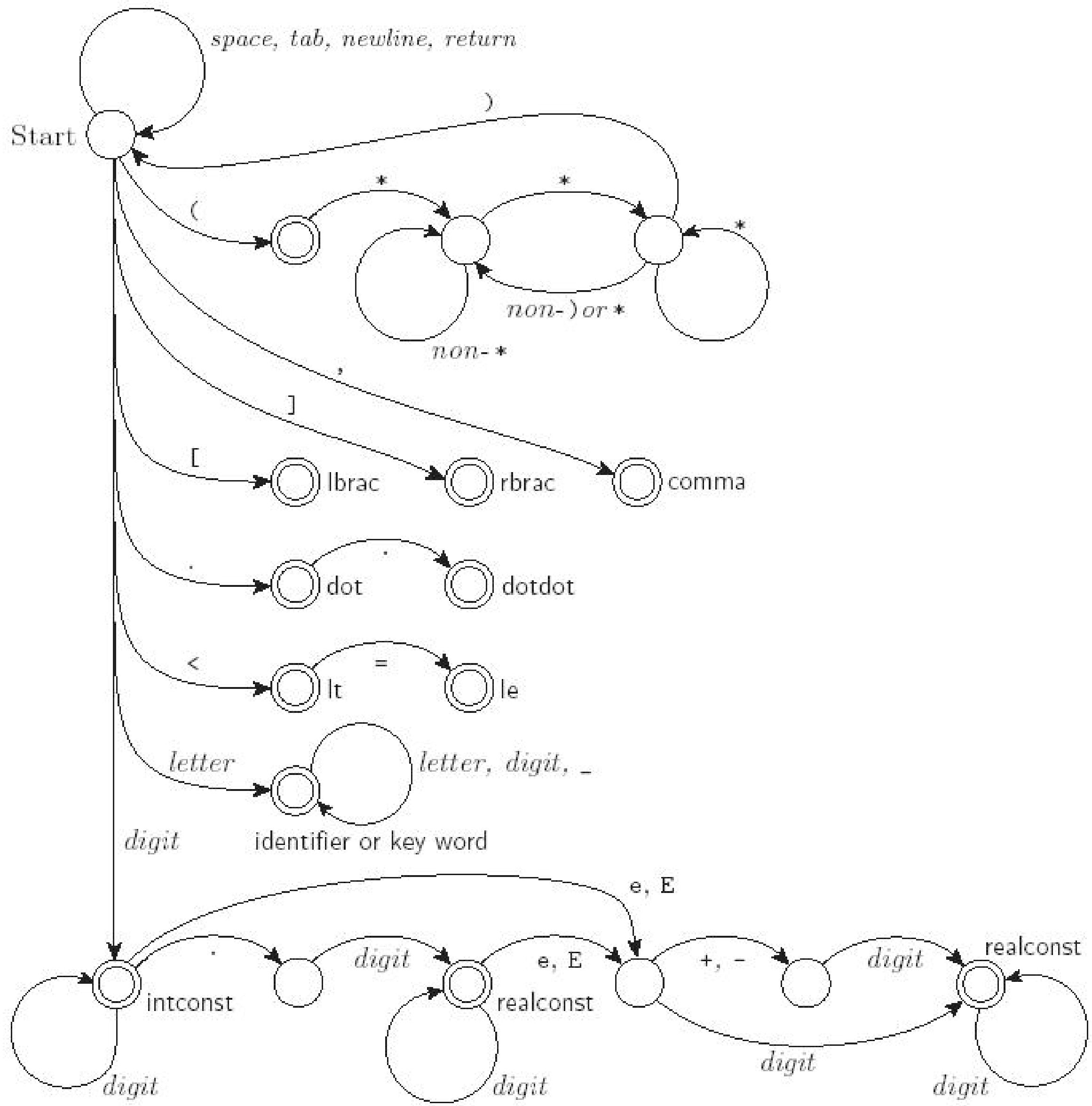
NFA for a | b



NFA for a*

Ken Thompson, *CACM*, 1968

Scanning as a Finite Automaton



The Whole Scanner Generator Process

- Overview:
  - Direct construction of a nondeterministic finite automaton (NFA) to recognize a given RE
    - Easy to build in an algorithmic way
    - Requires ε-transitions to combine regular subexpressions
  - Construct a deterministic finite automaton (DFA) to simulate the NFA
    - Use a set-of-states construction
  - Minimize the number of states in the DFA
    - Hopcroft state minimization algorithm
  - Generate the scanner code
    - Additional specifications needed for the actions

Optional, but worthwhile

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!

However, 3 Major Ways to Build Scanners

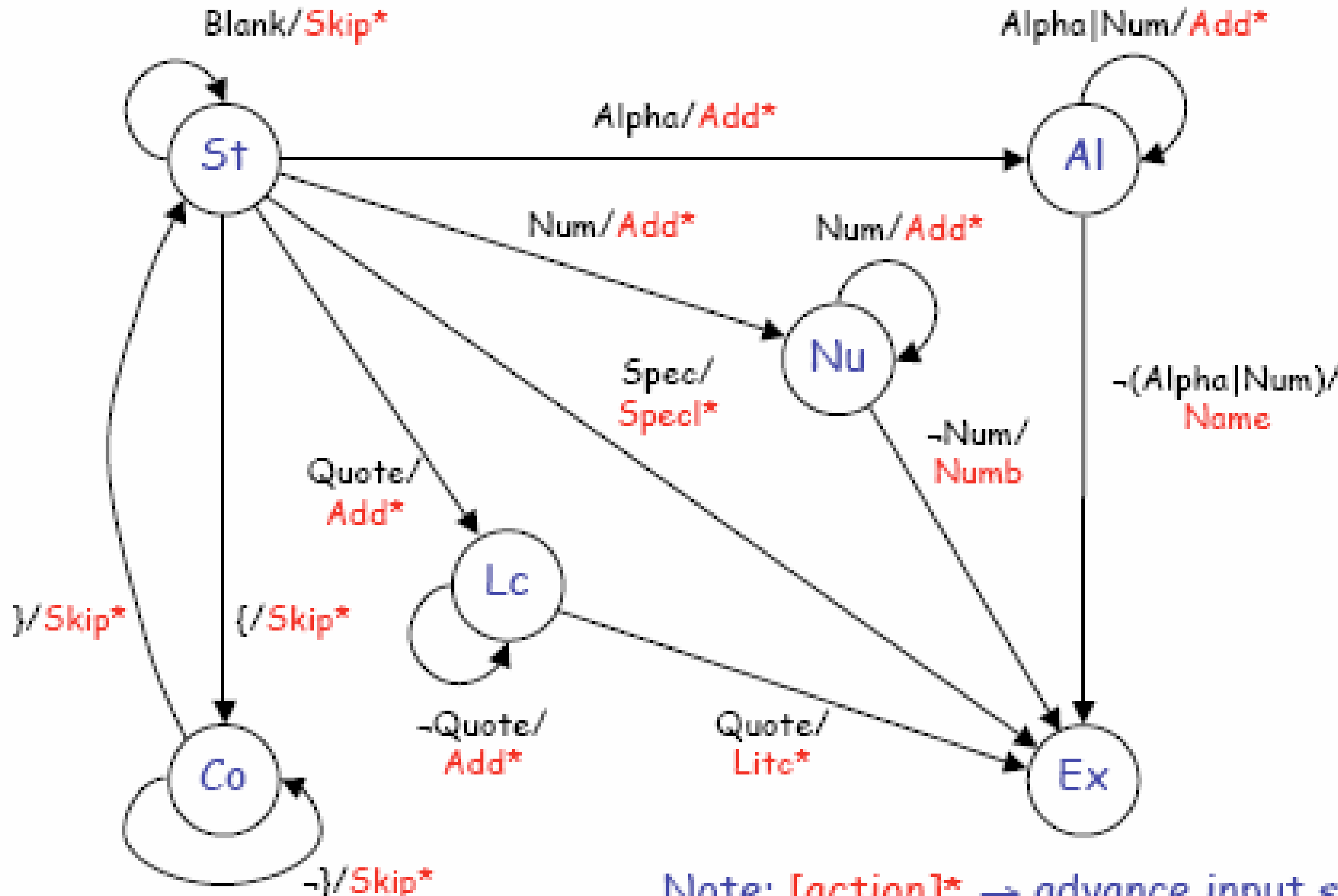
- ad-hoc
- semi-mechanical pure DFA
  (usually realized as nested case statements)
- table-driven DFA

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

A Semi-mechanical DFA Way

- Lexical Analysis Strategy: Simulation of Finite Automaton
  - States, characters, actions
  - State transition δ(state,charclass) determines next state
- Next character function
  - Reads next character into buffer
  - Computes *character class* by fast table lookup
- Transitions from state to state
  - Current state and next character determine (via δ)
    - Next state and action to be performed
    - Some actions *preload* next character
- Identifiers distinguished from keywords by hashed lookup
  - This differs from EAC advice (discussion later)
  - Permits translation of identifiers into <type, symbol_index>
    - Keywords each get their own type

A Lexical Analysis Example

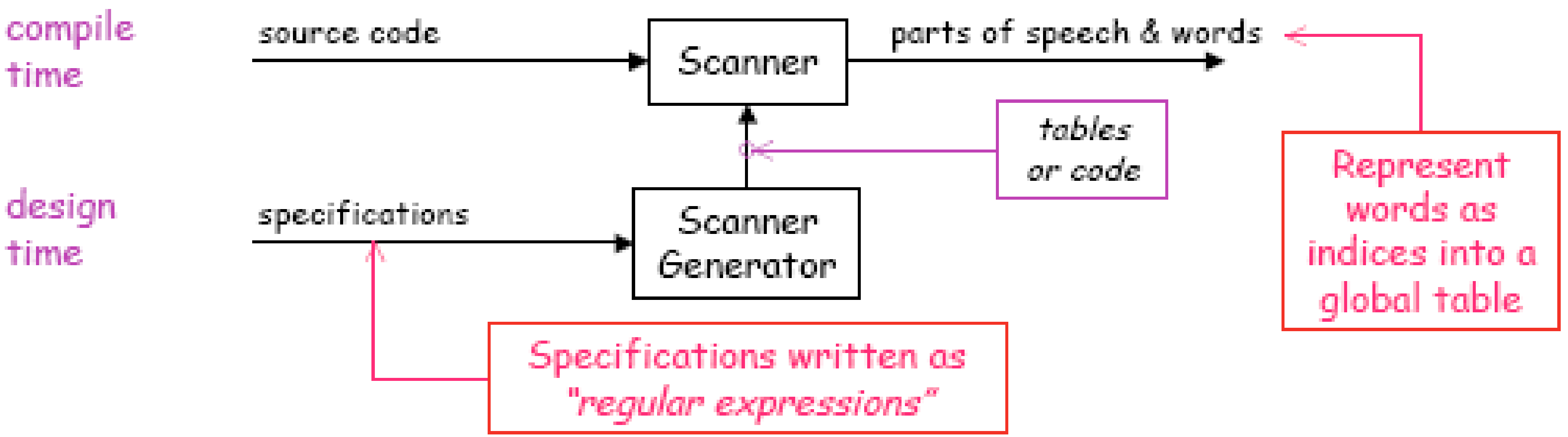


Note: [action]* → advance input stream

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);
}
```

The Scanner Generator Way



More on the Scanner Generator on Thursday…

Since the scanner is the only phase to touch the input source file, what else does it need to do?

Form of a Lex/Flex Spec File

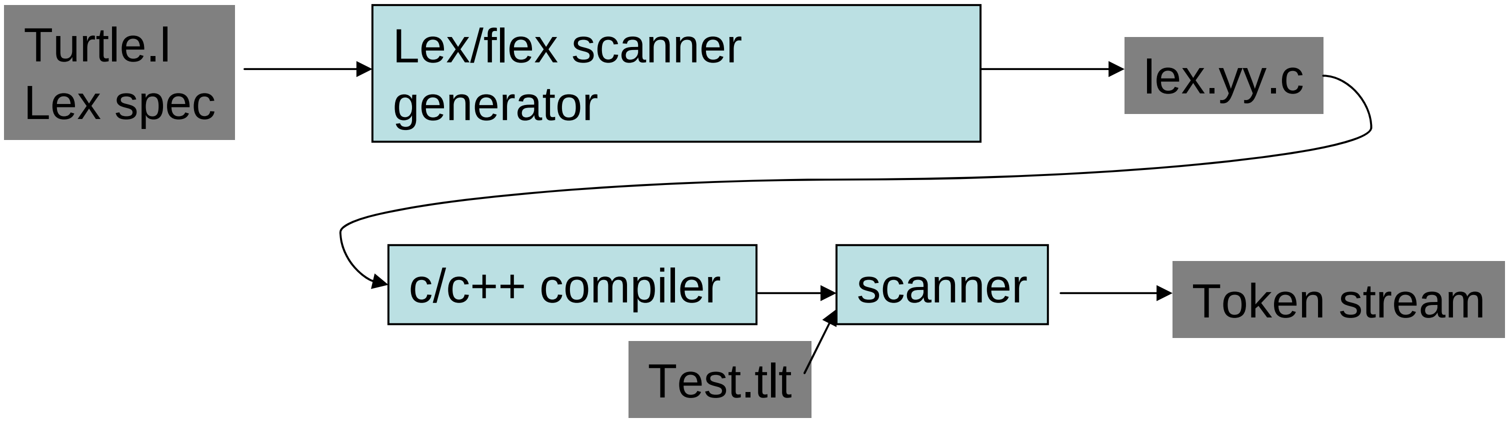
```
Definitions/declarations used for re clarity
%%
Reg exp0   {action0}  // translation rules to be
Reg exp1   {action1}   // converted to scanner
…                      …
%%
Auxiliary functions to be copied directly
```

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
- yyleng – length of lexeme matched
- yytext – points to start of lexeme



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

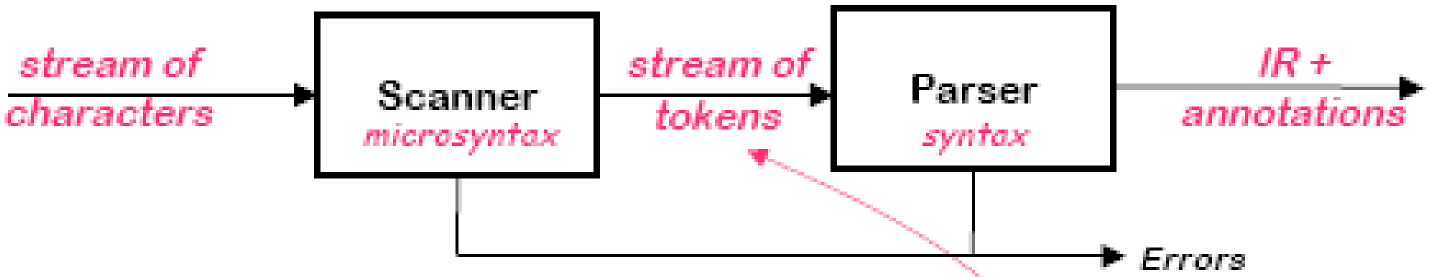
```
%%

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;



}
}
}
```

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

The scanner is responsible for:

- tokenizing source
- removing comments
- (often) dealing with *pragmas* (i.e., significant 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>

Slide From Keith Cooper, Rice