Flex & Bison Introduction

School of Computer Science & Technology

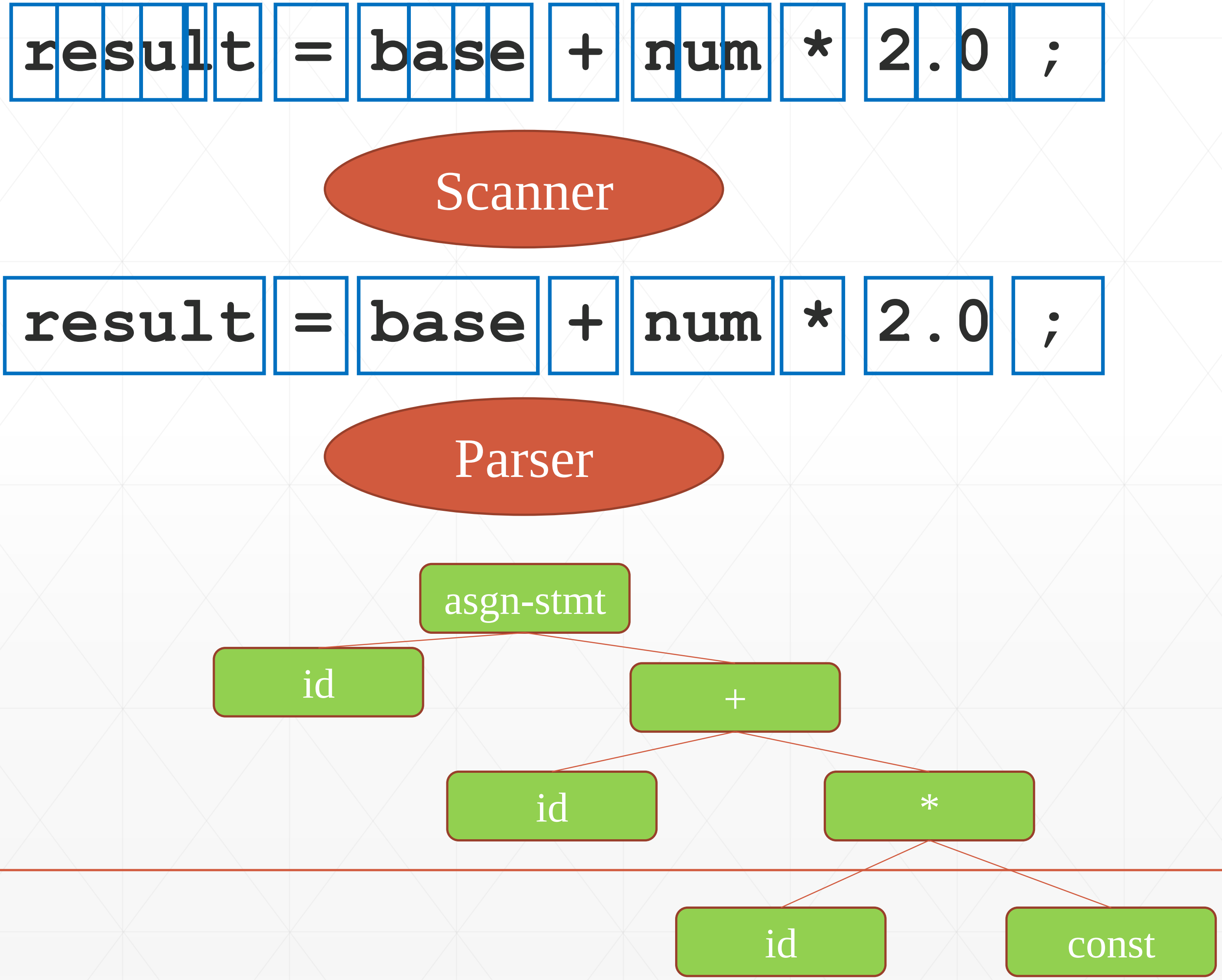
Thanks

- Most of the content is from or based off of information from here
  - Larry Ruzzo, University of Washington
  - Aaron Myles Landwehr, University of Delaware

Content

- Overview
- Flex
- Bison

Overview



r e s u l t = b a s e + n u m * 2 . 0 ;

Scanner

result = base + num * 2.0 ;

Parser

asgn-stmt

id

+

id

*

id

const

Overview

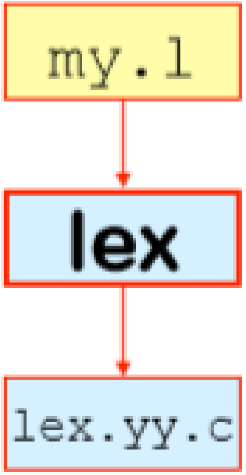
- Two tools
  - Lexical Tokens and their Order of Processing (Lex)
  - Context Free Grammar for LALR(1) (Yacc)
- Similar tools
  - Lex and Yacc – Earliest Days of Unix Minicomputers
  - Flex and Bison – From GNU
  - JFlex - Fast Scanner Generator for Java
  - BYacc/J – Berkeley
  - CUP, ANTRL, PCYACC, …
  - PCLEX and PCYACC from Abacus

Overview

- Lex
  - Generator of lexical analyzers
  - Written by Mike Lesk and Eric Schmidt
  - Isn’t used anymore
- Flex(fast lexical analyzer generator)
  - Free and open source alternative

Overview

- Input
  - Regular expression defining “tokens”
  - Fragments of C declarations & code
- Output
  - A C program “lex.yy.c”
- Use
  - Compile & link with your main()
  - Calls to yylex() read chars & return successive tokens

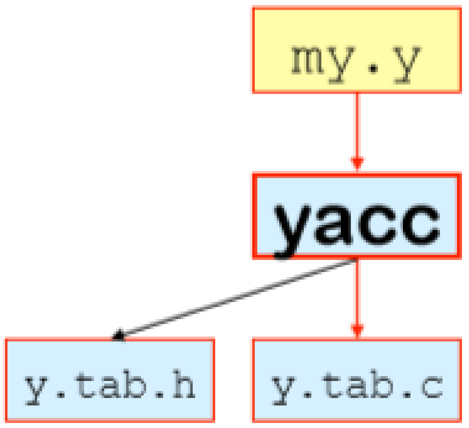


Overview

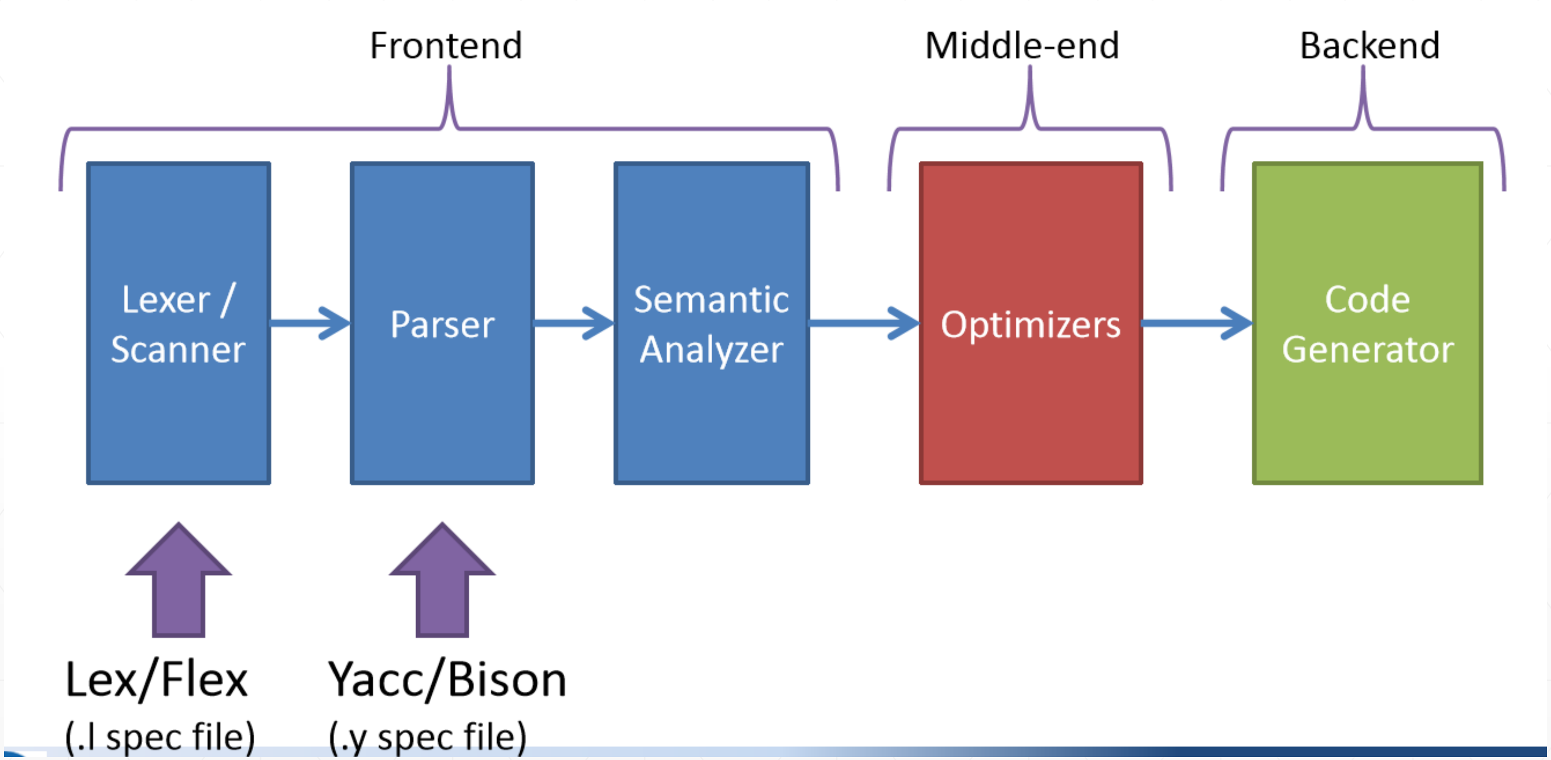
- Yacc
  - Syntactic analyzer generator
  - Requires a lexical analyzer
- Bison
  - Free and open source alternative
  - We will use this

Overview

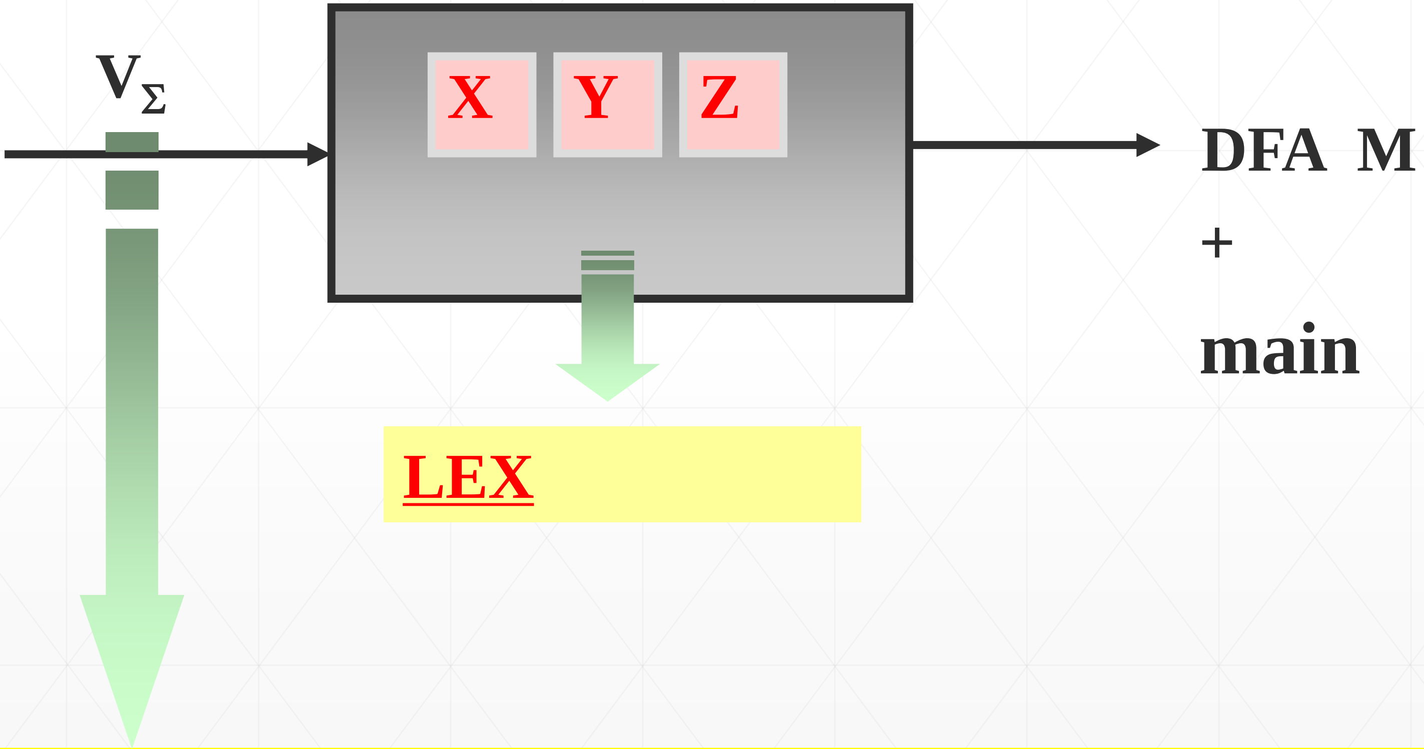
- Input
  - A context-free grammar
  - Fragments of C declarations & code
- Output
  - A C program & some header files
- Use
  - Compile & link it with main()
  - Call yyparse() to parse input source
  - yyparse() calls yylex() to get successive tokens



Overview


Frontend
Middle-end
Backend
Lexer / Scanner
Parser
Semantic Analyzer
Optimizers
Code Generator
Lex/Flex
(.l spec file)
Yacc/Bison
(.y spec file)

Lex/Flex – A Lexical Analyzer Generator



V_Σ

X Y Z

DFA M

+

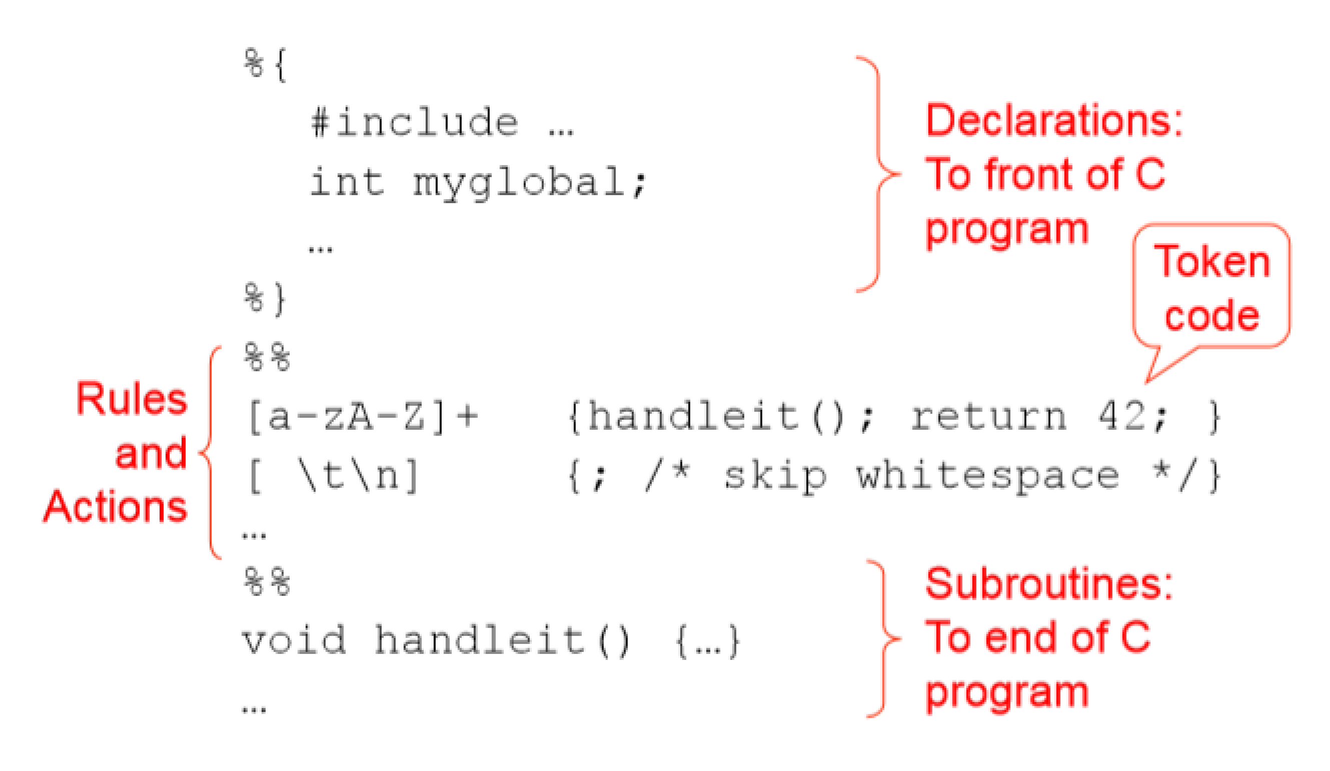
main

LEX

lex specification

Lex/Flex – A Lexical Analyzer Generator

- Input: example.l



```
%{
   #include …
   int myglobal;
   …
%}
%%
[a-zA-Z]+    {handleit(); return 42; }
[ \t\n]      {; /* skip whitespace */}
…
%%
void handleit() {…}
…
```

Declarations: To front of C program

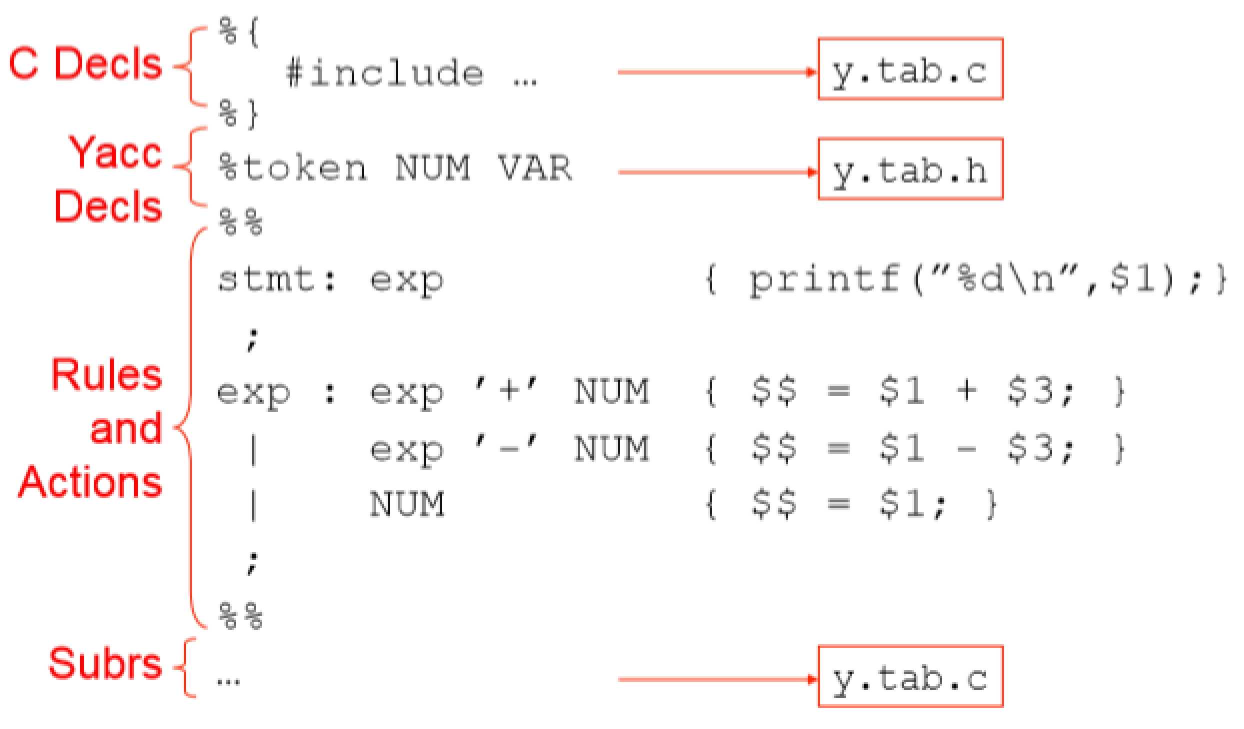
Token code

Rules and Actions

Subroutines: To end of C program

Yacc/Bison – A Syntax Analyzer Generator

- Input: example.y

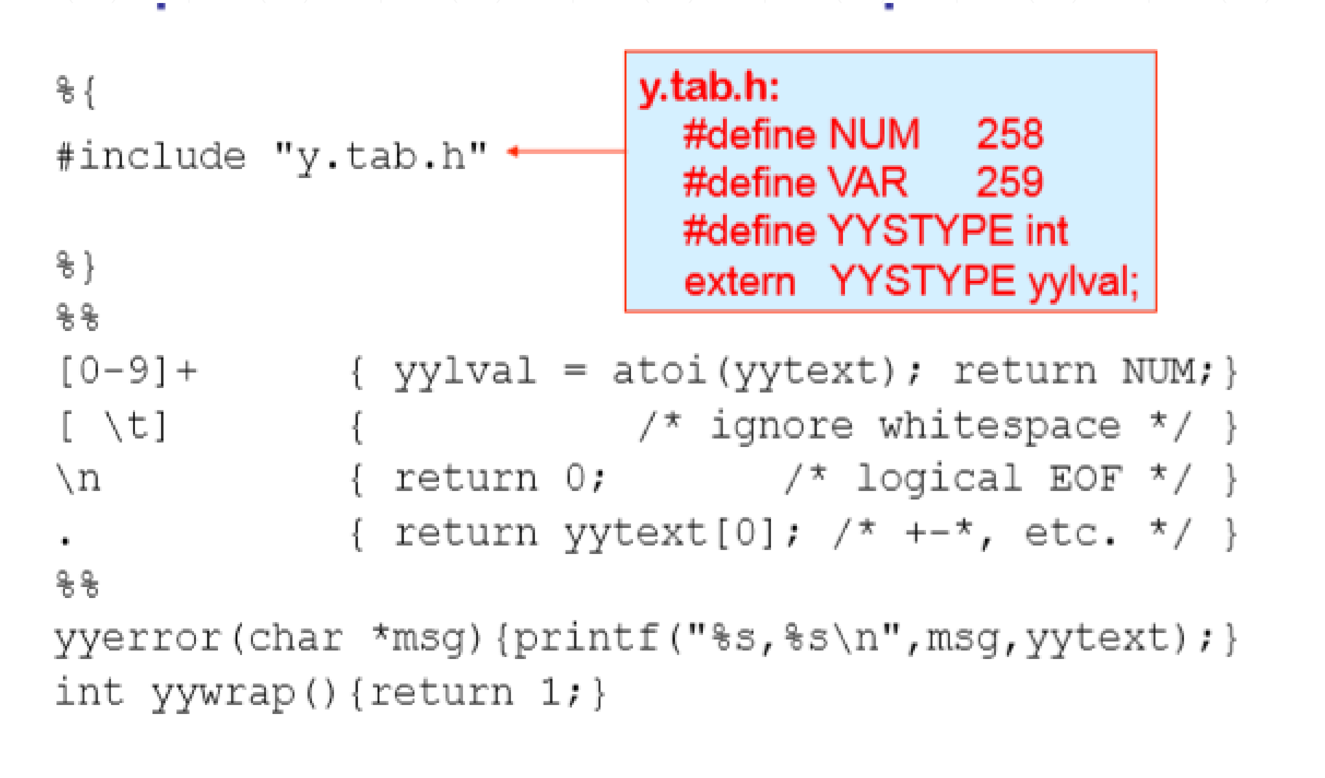


```
%{
   #include …
%}
%token NUM VAR
%%
stmt: exp                { printf("%d\n",$1);}
  ;
exp : exp '+' NUM        { $$ = $1 + $3; }
  |     exp '-' NUM      { $$ = $1 - $3; }
  |     NUM              { $$ = $1; }
  ;
%%
…
```

S → E
E → E+n | E-n | n

Lex & Yacc / Flex & Bison

- Input: example.l



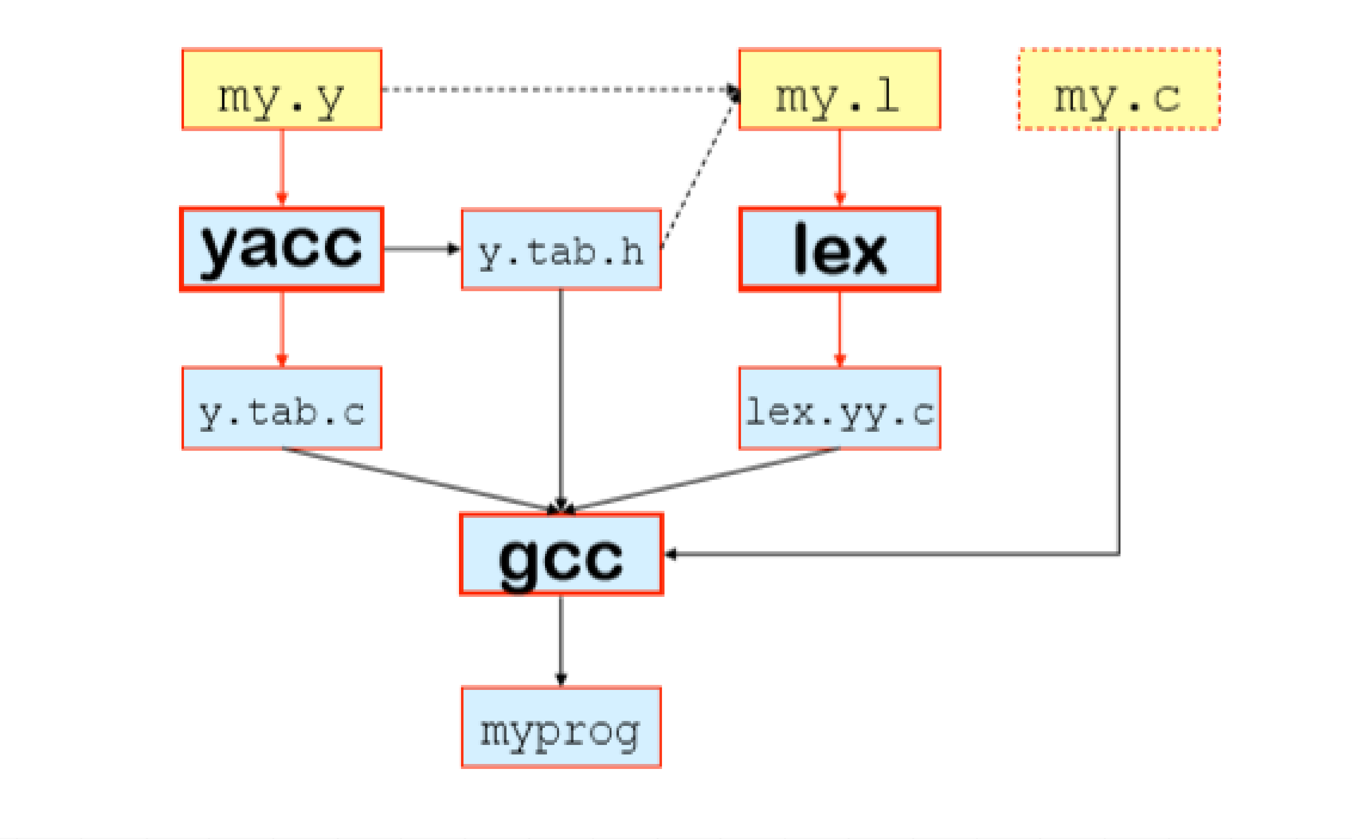
```
%{
#include "y.tab.h"

%}
%%
[0-9]+        { yylval = atoi(yytext); return NUM;}
[ \t]         {             /* ignore whitespace */ }
\n            { return 0;          /* logical EOF */ }
.             { return yytext[0]; /* +-*, etc. */ }
%%
yyerror(char *msg){printf("%s,%s\n",msg,yytext);}
int yywrap(){return 1;}
```

```
y.tab.h:
   #define NUM     258
   #define VAR     259
   #define YYSTYPE int
   extern   YYSTYPE yylval;
```

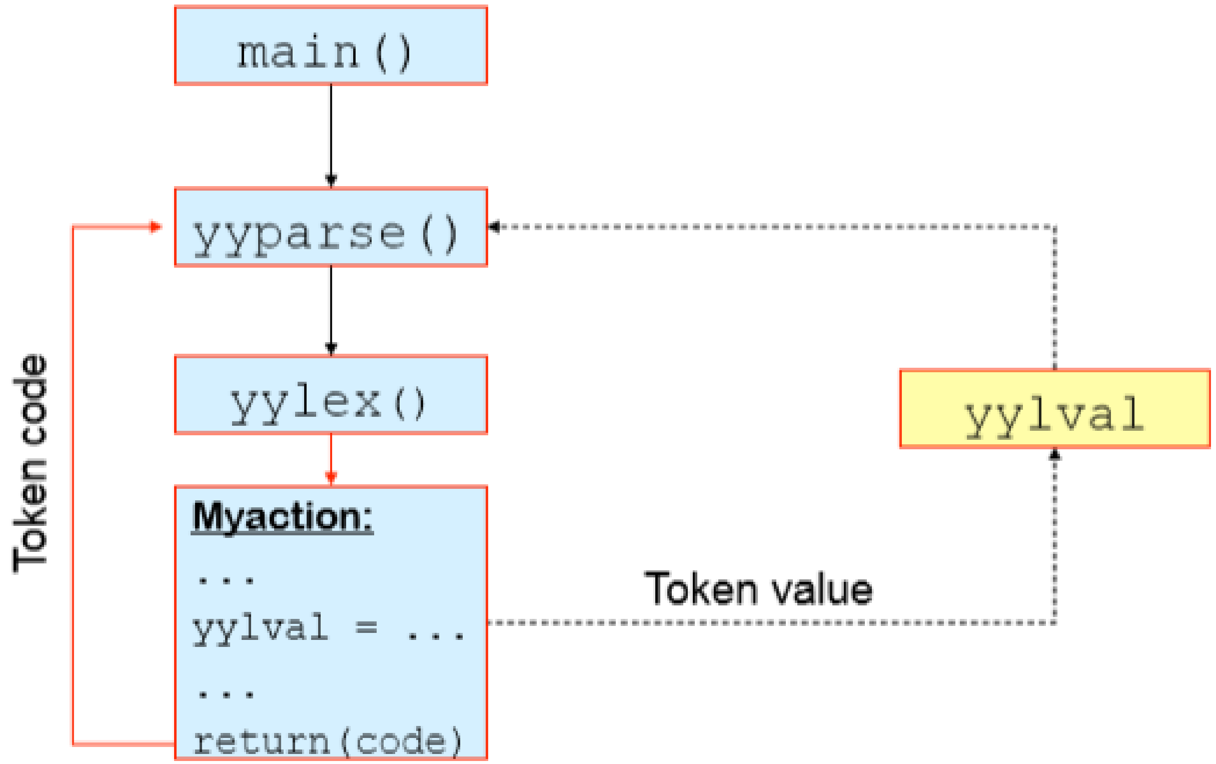
Lex & Yacc / Flex & Bison

- Interface



Lex & Yacc / Flex & Bison

- Runtime Interface



main()

yyparse()

yylex()

yylval

Token code

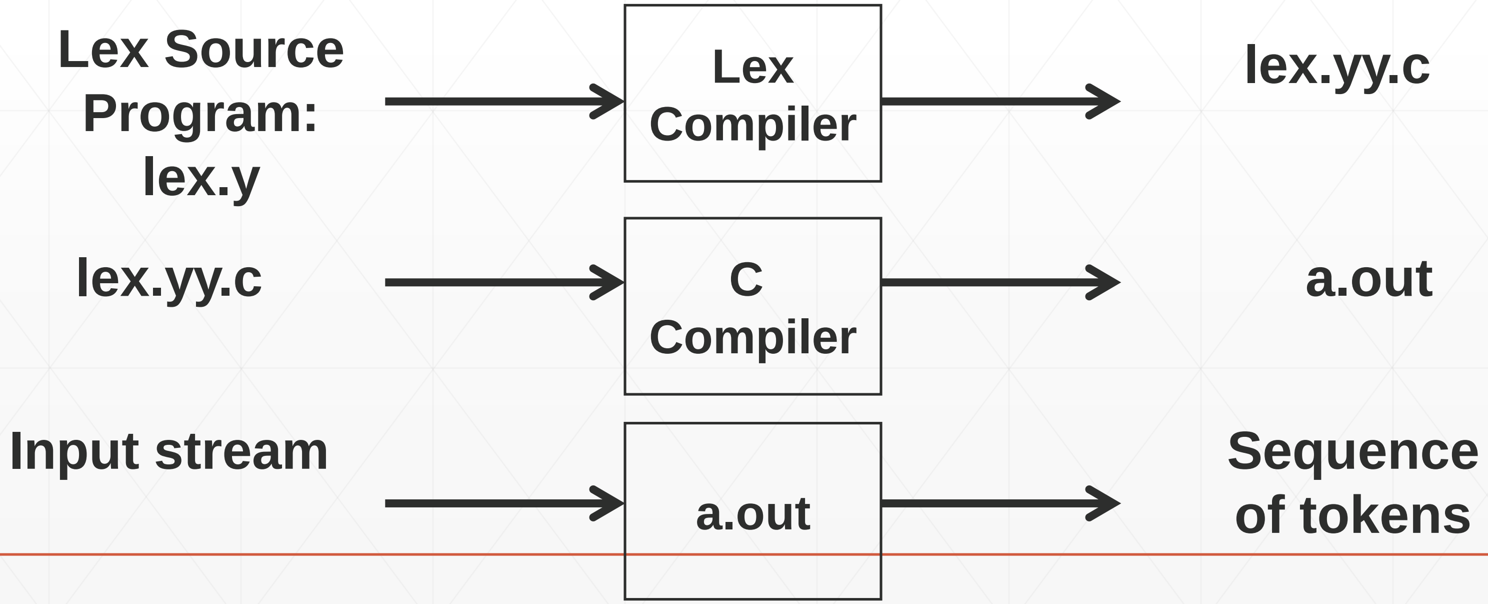
Token value

Myaction:

```
...
yylval = ...
...
return(code)
```

Lex/Flex – A Lexical Analyzer Generator

- A Unix Utility from early 1970s
- A Compiler that takes as source a specification for
  - Tokens/Patterns of a Language
  - Generates a "C" Lexical Analyzer Program
- Pictorially:



Lex Source Program: lex.y

Lex Compiler

lex.yy.c

lex.yy.c

C Compiler

a.out

Input stream

a.out

Sequence of tokens

Lex/Flex – A Lexical Analyzer Generator

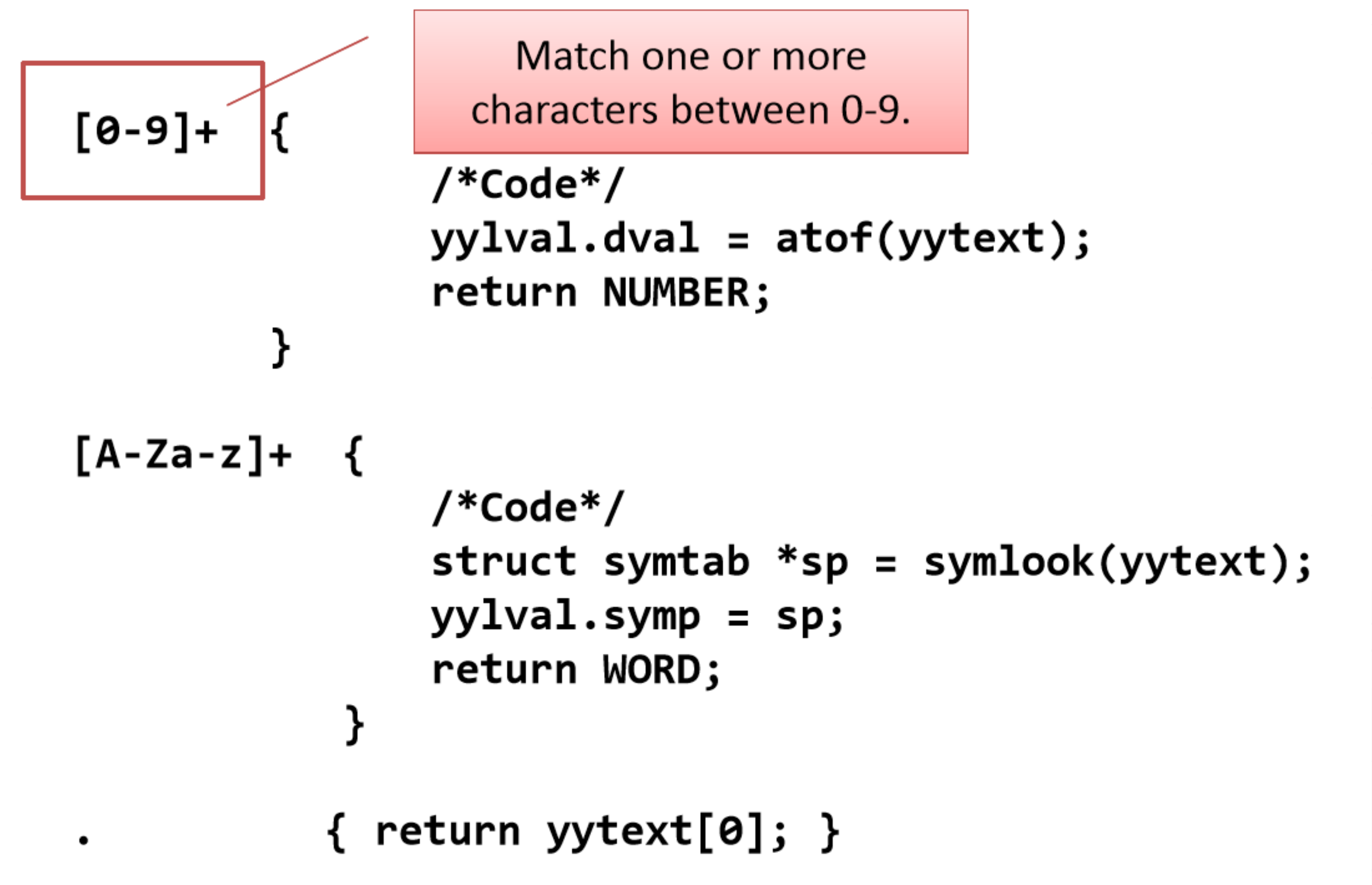
- Declarations:
  - Defs, Constants, Types, #includes, etc. that can Occur in a C Program
  - Regular Definitions (expressions)
- Translation Rules:
  - Pairs of (Regular Expression, Action)
  - Informs Lexical Analyzer of Action when Pattern is Recognized
- Auxiliary Procedures:
  - Designer Defined C Code
  - Can Replace System Calls
- See Also
  - http://www.cs.fsu.edu/~langley/COP4342-2006-Fall/17-programdevel04.pdf
  - http://alumni.cs.ucr.edu/~lgao/teaching/flex.html

Lex.y File Format:
DECLARATIONS
%%
TRANSLATION RULES
%%
AUXILIARY PROCEDURES

Lex/Flex – A Lexical Analyzer Generator

- char *yytext;
  - Pointer to current lexeme terminated by ‘\0’
- int yylen;
  - Number of chacters in yytex but not ‘\0’
- yylval:
  - Global variable through which the token value can be returned to Yacc
  - Parser (Yacc) can access yylval, yylen, and yytext
- How are these used?
  - Consider Integer Tokens:
  - yylval = ascii_to_integer (yytext);
  - Conversion from String to actual Integer Value

Lex/Flex – A Lexical Analyzer Generator



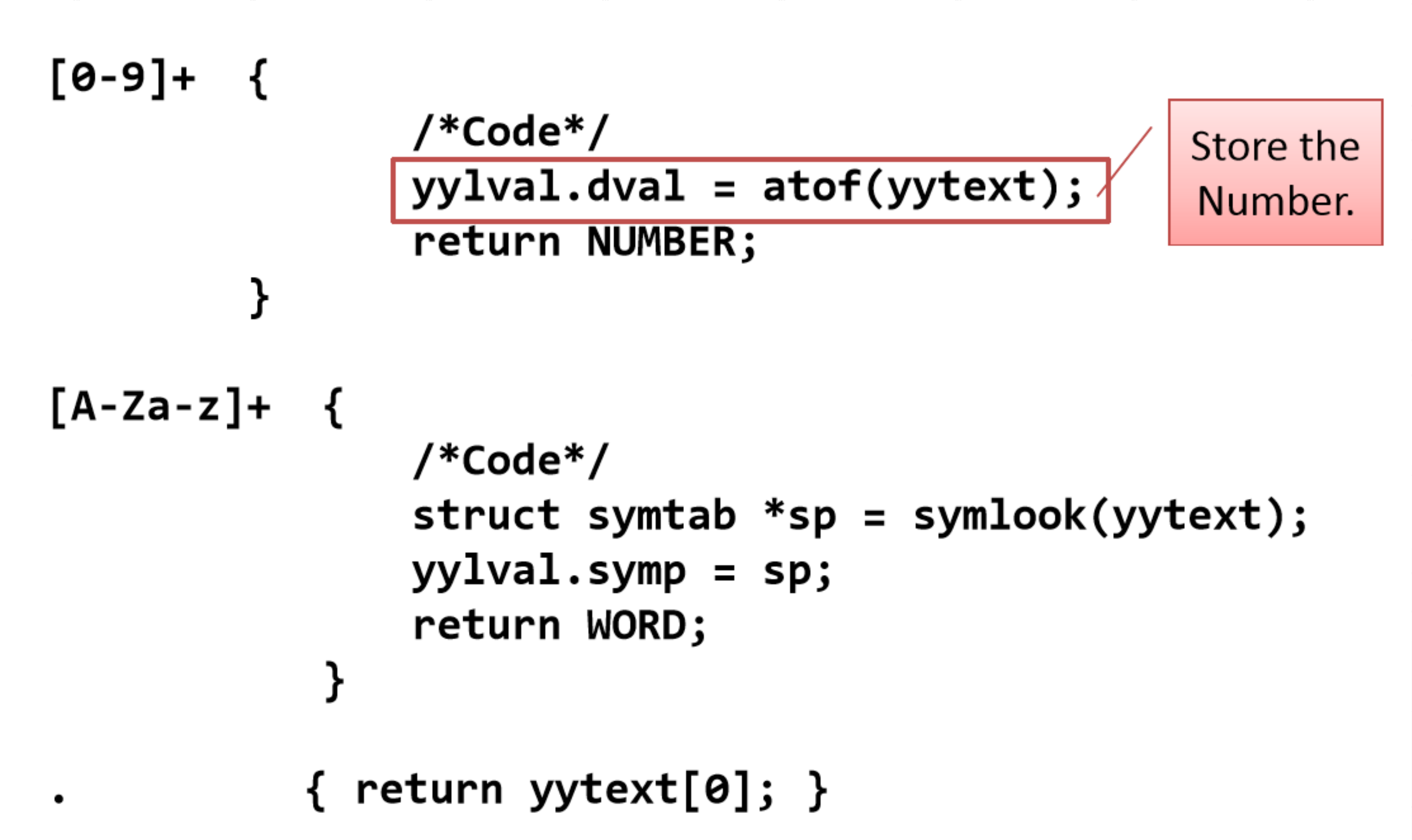
Match one or more characters between 0-9.

```
[0-9]+    {
                 /*Code*/
                 yylval.dval = atof(yytext);
                 return NUMBER;
          }

[A-Za-z]+   {
                 /*Code*/
                 struct symtab *sp = symlook(yytext);
                 yylval.symp = sp;
                 return WORD;
            }

.          { return yytext[0]; }
```

Lex/Flex – A Lexical Analyzer Generator



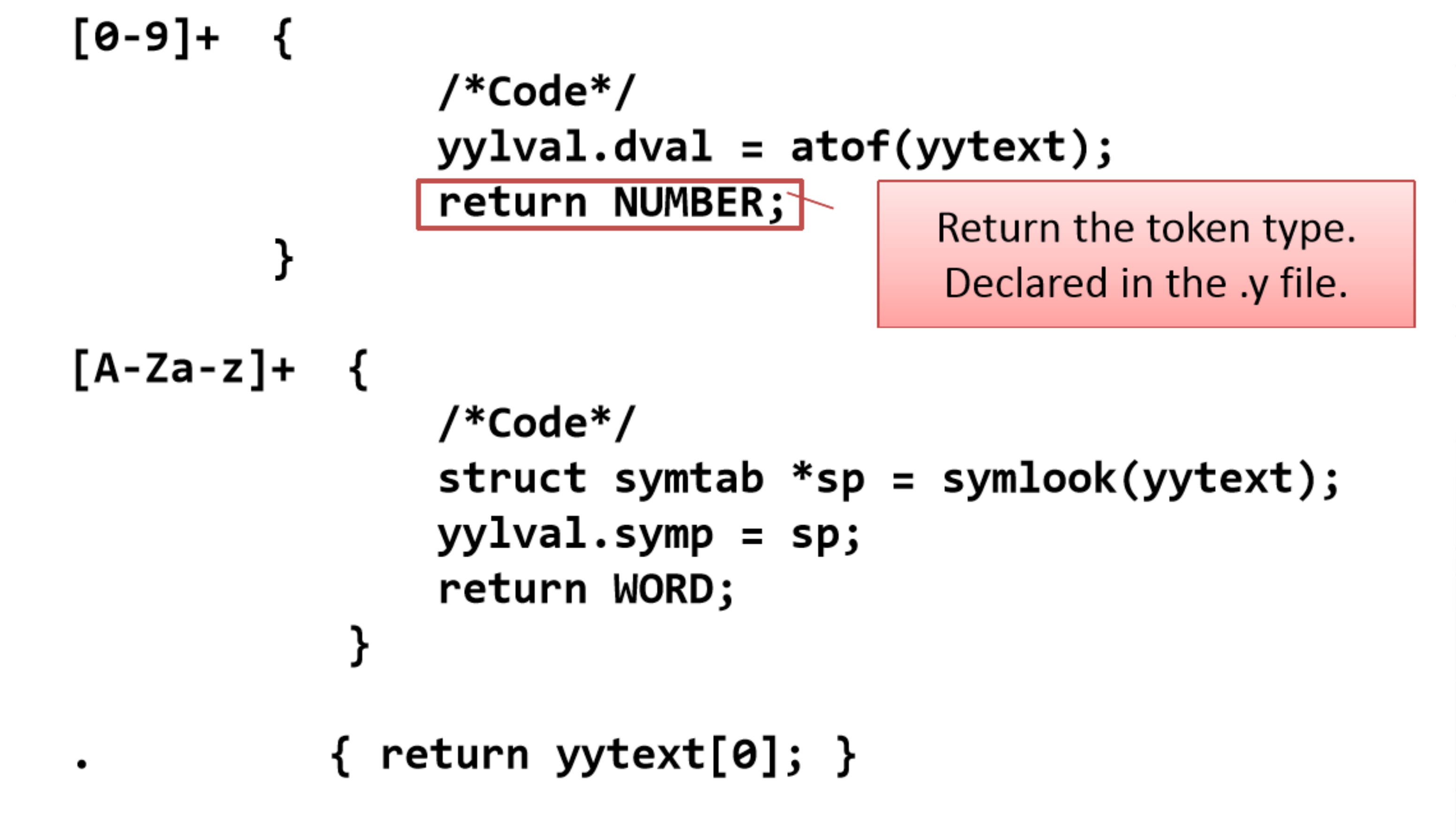
```
[0-9]+  {
               /*Code*/
               yylval.dval = atof(yytext);
               return NUMBER;
        }

[A-Za-z]+  {
               /*Code*/
               struct symtab *sp = symlook(yytext);
               yylval.symp = sp;
               return WORD;
           }

.          { return yytext[0]; }
```

Store the Number.

Lex/Flex – A Lexical Analyzer Generator



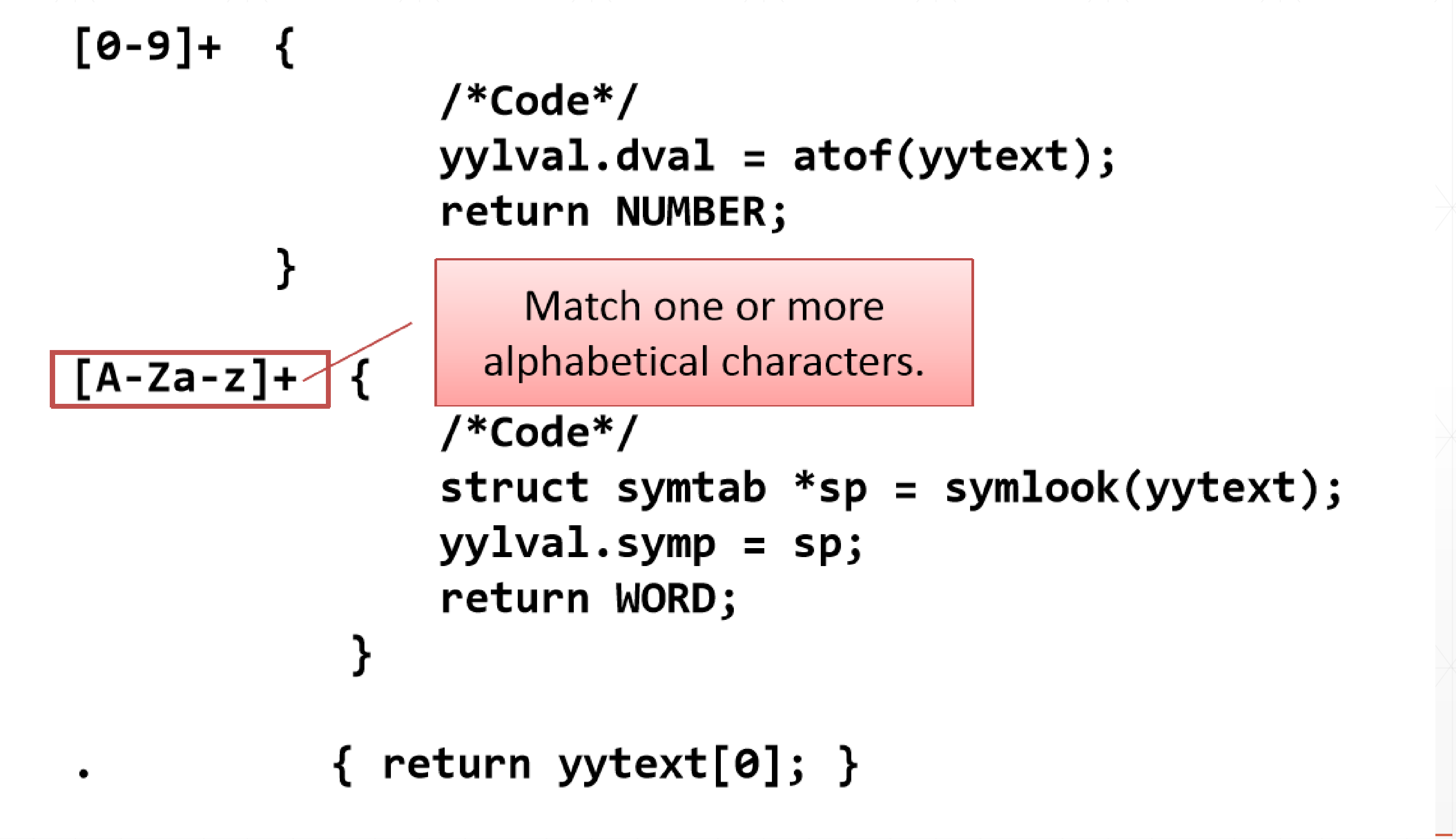
```
[0-9]+  {
              /*Code*/
              yylval.dval = atof(yytext);
              return NUMBER;
        }

[A-Za-z]+  {
              /*Code*/
              struct symtab *sp = symlook(yytext);
              yylval.symp = sp;
              return WORD;
           }

.         { return yytext[0]; }
```

Return the token type.
Declared in the .y file.

Lex/Flex – A Lexical Analyzer Generator



```
[0-9]+  {
              /*Code*/
              yylval.dval = atof(yytext);
              return NUMBER;
        }

[A-Za-z]+  {
              /*Code*/
              struct symtab *sp = symlook(yytext);
              yylval.symp = sp;
              return WORD;
           }

.          { return yytext[0]; }
```

Match one or more alphabetical characters.

Lex/Flex – A Lexical Analyzer Generator

```
[0-9]+  {
              /*Code*/
              yylval.dval = atof(yytext);
              return NUMBER;
        }

[A-Za-z]+  {
              /*Code*/
              struct symtab *sp = symlook(yytext);
              yylval.symp = sp;
              return WORD;
           }

.         { return yytext[0]; }
```

Store the text.

Lex/Flex – A Lexical Analyzer Generator

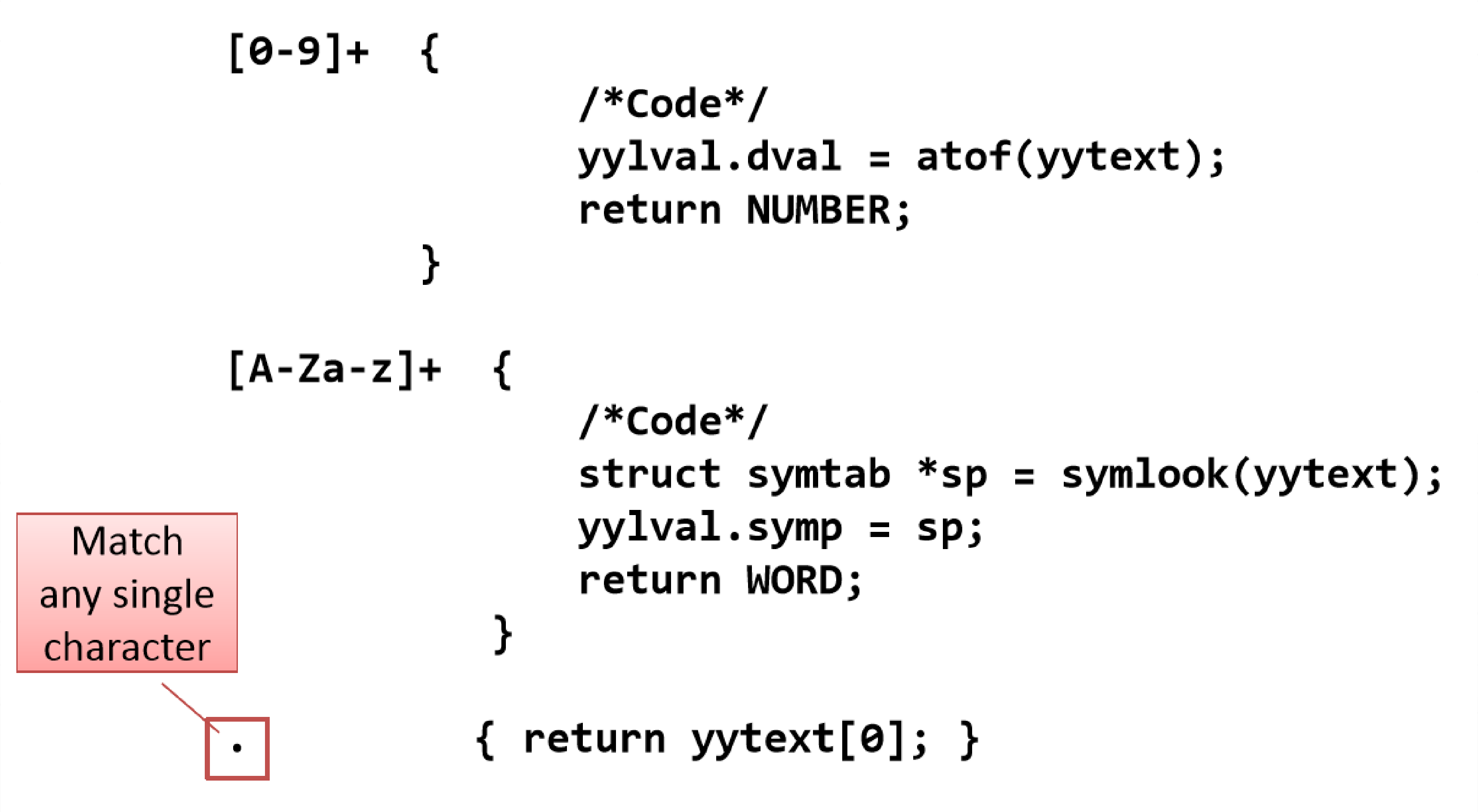
```
[0-9]+  {
              /*Code*/
              yylval.dval = atof(yytext);
              return NUMBER;
        }

[A-Za-z]+  {
              /*Code*/
              struct symtab *sp = symlook(yytext);
              yylval.symp = sp;
              return WORD;
           }

.         { return yytext[0]; }
```

Return the token type.
Declared in the .y file.

Lex/Flex – A Lexical Analyzer Generator



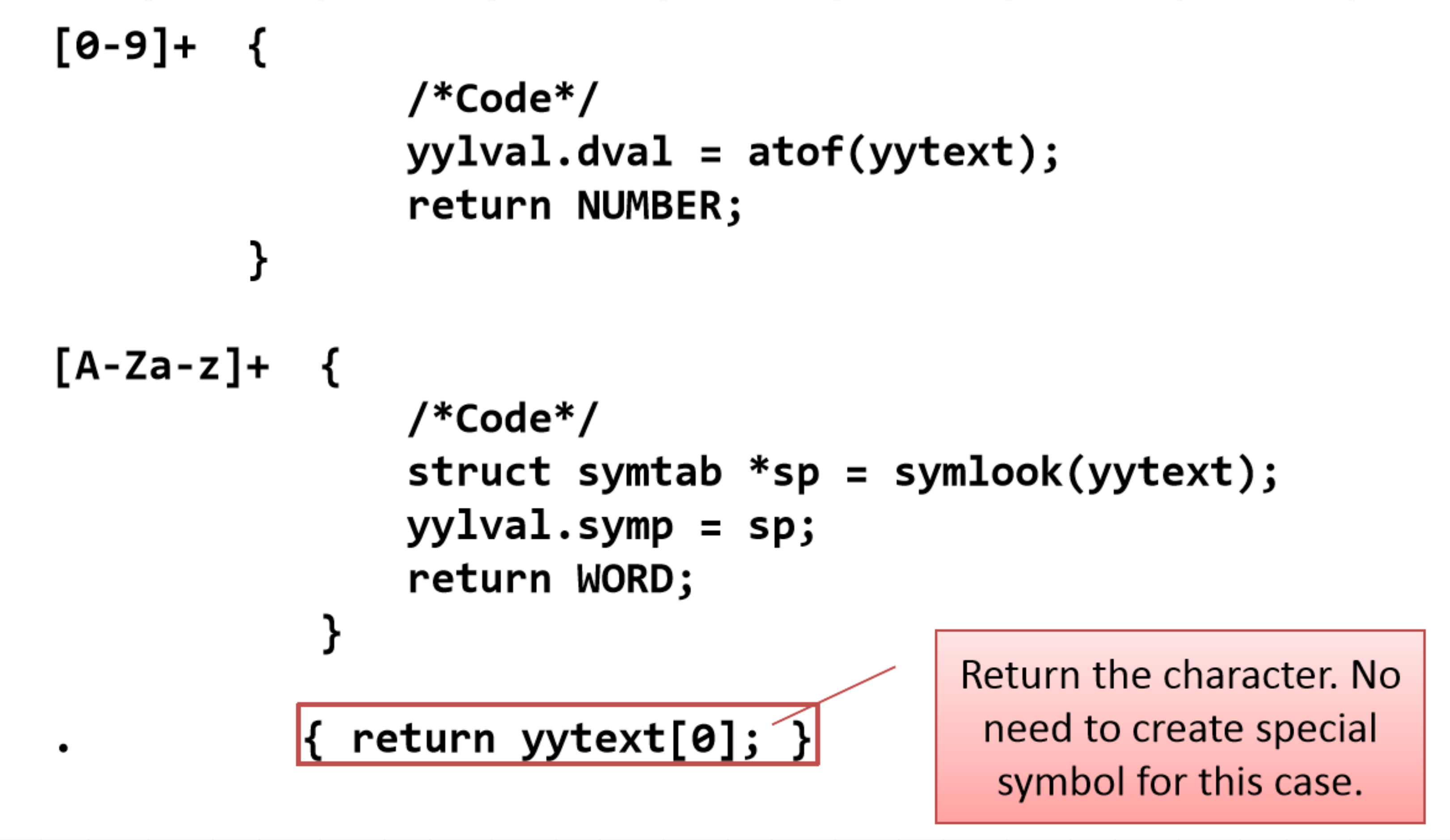
```
[0-9]+  {
               /*Code*/
               yylval.dval = atof(yytext);
               return NUMBER;
        }

[A-Za-z]+  {
               /*Code*/
               struct symtab *sp = symlook(yytext);
               yylval.symp = sp;
               return WORD;
           }

.         { return yytext[0]; }
```

Match any single character

Lex/Flex – A Lexical Analyzer Generator



```
[0-9]+  {
                /*Code*/
                yylval.dval = atof(yytext);
                return NUMBER;
        }

[A-Za-z]+  {
                /*Code*/
                struct symtab *sp = symlook(yytext);
                yylval.symp = sp;
                return WORD;
           }

.           { return yytext[0]; }
```

Return the character. No need to create special symbol for this case.

```
%{
#define  T_IDENTIFIER  300
#define  T_INTEGER     301
#define  T_REAL        302
#define  T_STRING      303
#define  T_ASSIGN      304
#define  T_ELSE        305
#define  T_IF          306
#define  T_THEN        307
#define  T_EQ          308
#define  T_LT          309
#define  T_NE          310
#define  T_GE          311
#define  T_GT          312
%}
```

User Defined Values to Each Token (else lex will assign)

```
letter            [a-zA-Z]
digit             [0-9]
ws                [ \t\n]+
id                [A-Za-z][A-Za-z0-9]*
comment           "(*"([^*]|\n|"*"+[^)])*"*"+")"
integer           [0-9]+/([^0-9]|"..")
real              [0-9]+"."[0-9]*([0-9]|"E"[+-]?[0-9]+)
string            \'([^']|\'\')*\'
%%
```

Regular Expression Rules for later token definitions

```
":="              {printf(" %s ", yytext);return(T_ASSIGN);}
"else"            {printf(" %s ", yytext);return(T_ELSE);}
```

Token Definitions

```
"then"                  {
#ifdef PRNTFLG
printf(" %s ", yytext);
#endif
            return(T_THEN);
            }
```

Conditional compilation action

```
"<="                    {printf(" %s ", yytext);return(T_EQ);}
"<"                     {printf(" %s ", yytext);return(T_LT);}
"<>"                    {printf(" %s ", yytext);return(T_NE);}
">="                    {printf(" %s ", yytext);return(T_GE);}
">"                     {printf(" %s ", yytext);return(T_GT);}

{id}                    {printf(" %s ", yytext);return(T_IDENTIFIER);}
{integer}       {printf(" %s ", yytext);return(T_INTEGER);}
{real}                  {printf(" %s ", yytext);return(T_REAL);}
{string}        {printf(" %s ", yytext);return(T_STRING);}
```

Token Definitions

```
{comment}               {/* T_COMMENT */}
{ws}                    {/* spaces, tabs, newlines */}
```

Discard

```
%%
yywrap(){return 0;}
```

EOF for input

```
main()
{
int i;
do {
  i = yylex();
} while (i!=0);
}
```

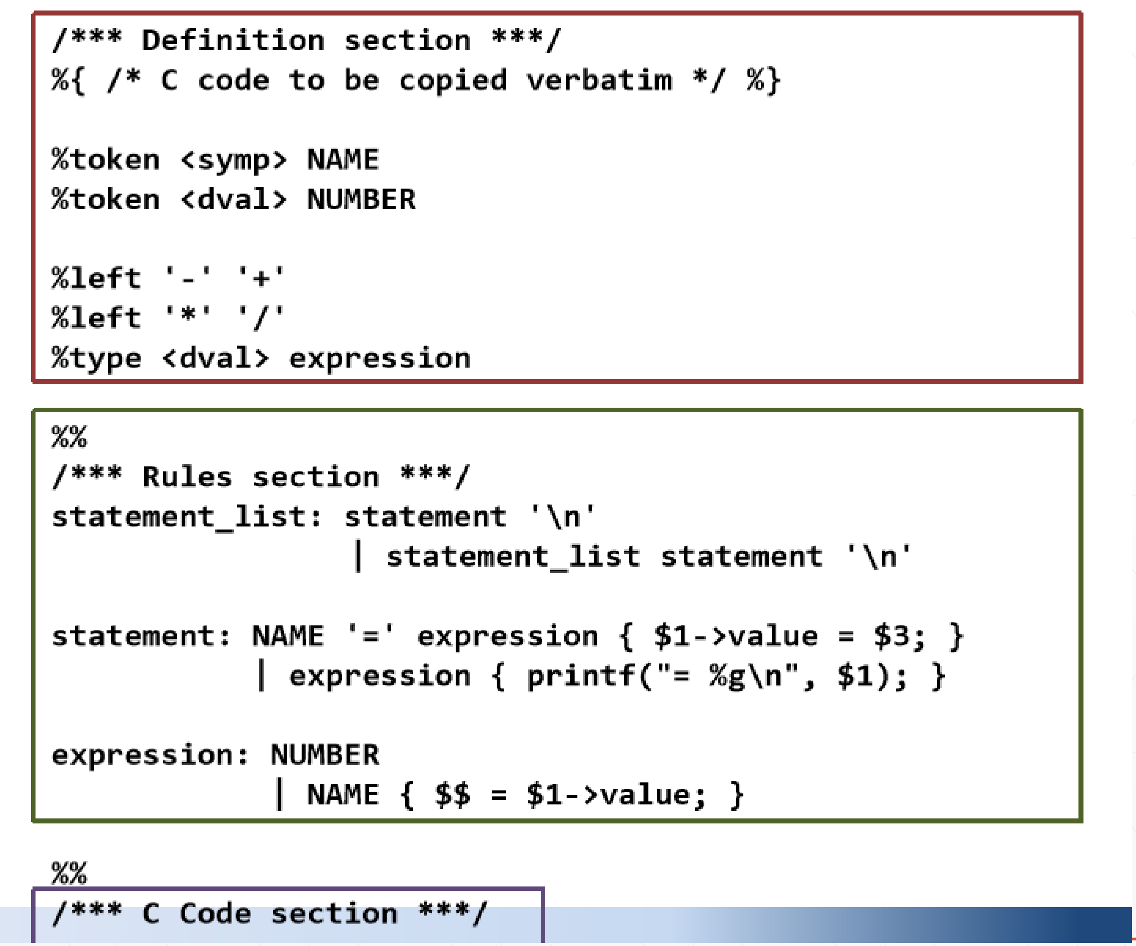
Three Variables:

yytext = “currenttoken”

yylen = 12

yylval = 300

Yacc/Bison – A Syntax Analyzer Generator



```
/*** Definition section ***/
%{ /* C code to be copied verbatim */ %}

%token <symp> NAME
%token <dval> NUMBER

%left '-' '+'
%left '*' '/'
%type <dval> expression
```

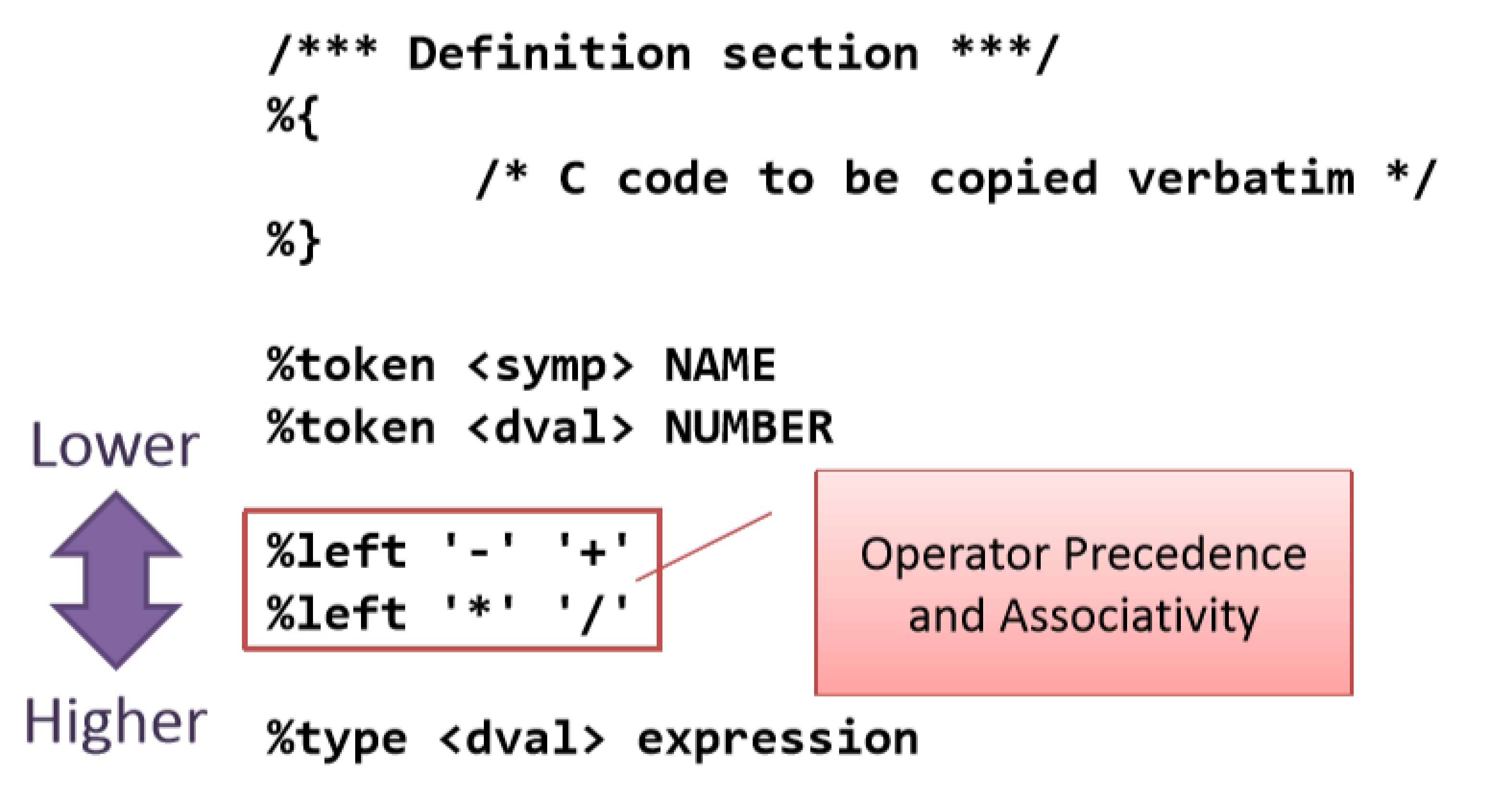
```
%%
/*** Rules section ***/
statement_list: statement '\n'
                | statement_list statement '\n'

statement: NAME '=' expression { $1->value = $3; }
           | expression { printf("= %g\n", $1); }

expression: NUMBER
            | NAME { $$ = $1->value; }
```

```
%%
/*** C Code section ***/
```

Yacc/Bison – A Syntax Analyzer Generator



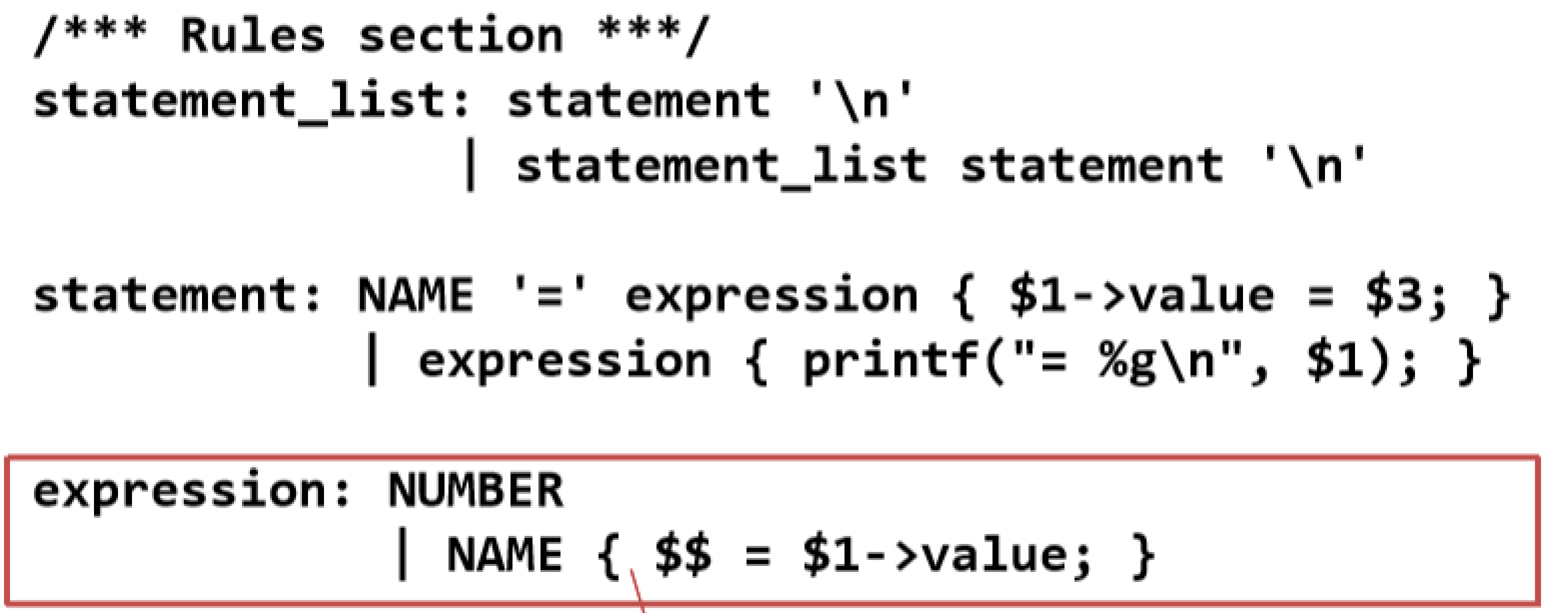
```
/*** Definition section ***/
%{
        /* C code to be copied verbatim */
%}

%token <symp> NAME
%token <dval> NUMBER

%left '-' '+'
%left '*' '/'

%type <dval> expression
```

Yacc/Bison – A Syntax Analyzer Generator

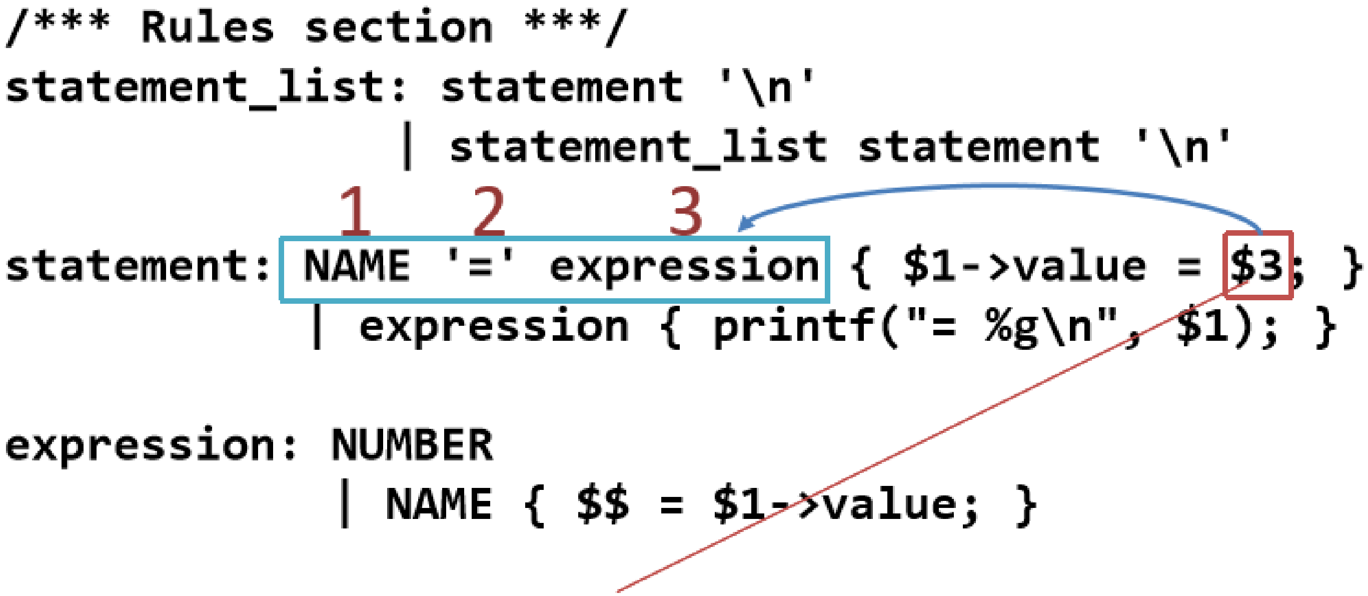


```
/*** Rules section ***/
statement_list: statement '\n'
              | statement_list statement '\n'

statement: NAME '=' expression { $1->value = $3; }
         | expression { printf("= %g\n", $1); }

expression: NUMBER
          | NAME { $$ = $1->value; }
```

This simply says that an expression is a **number** or a **name**.



```
/*** Rules section ***/
statement_list: statement '\n'
               | statement_list statement '\n'

statement: NAME '=' expression { $1->value = $3; }
           | expression { printf("= %g\n", $1); }

expression: NUMBER
            | NAME { $$ = $1->value; }
```

The numbers in the executable statement correspond to the tokens listed in the production. They are numbered in ascending order.