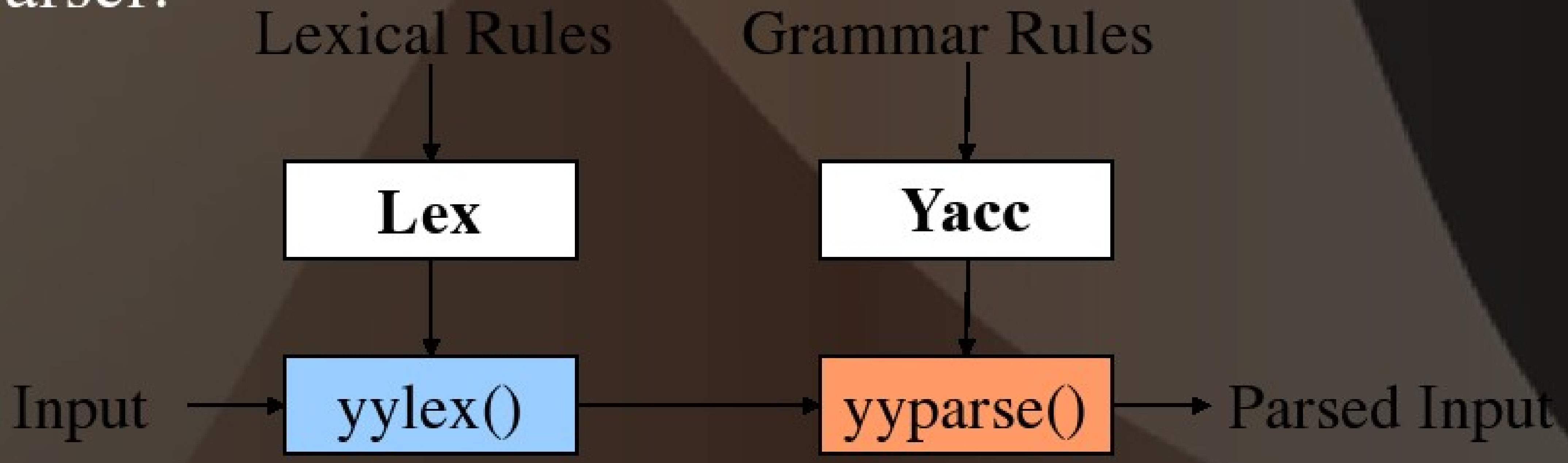
Lecture #20

Yet Another Compiler Compiler

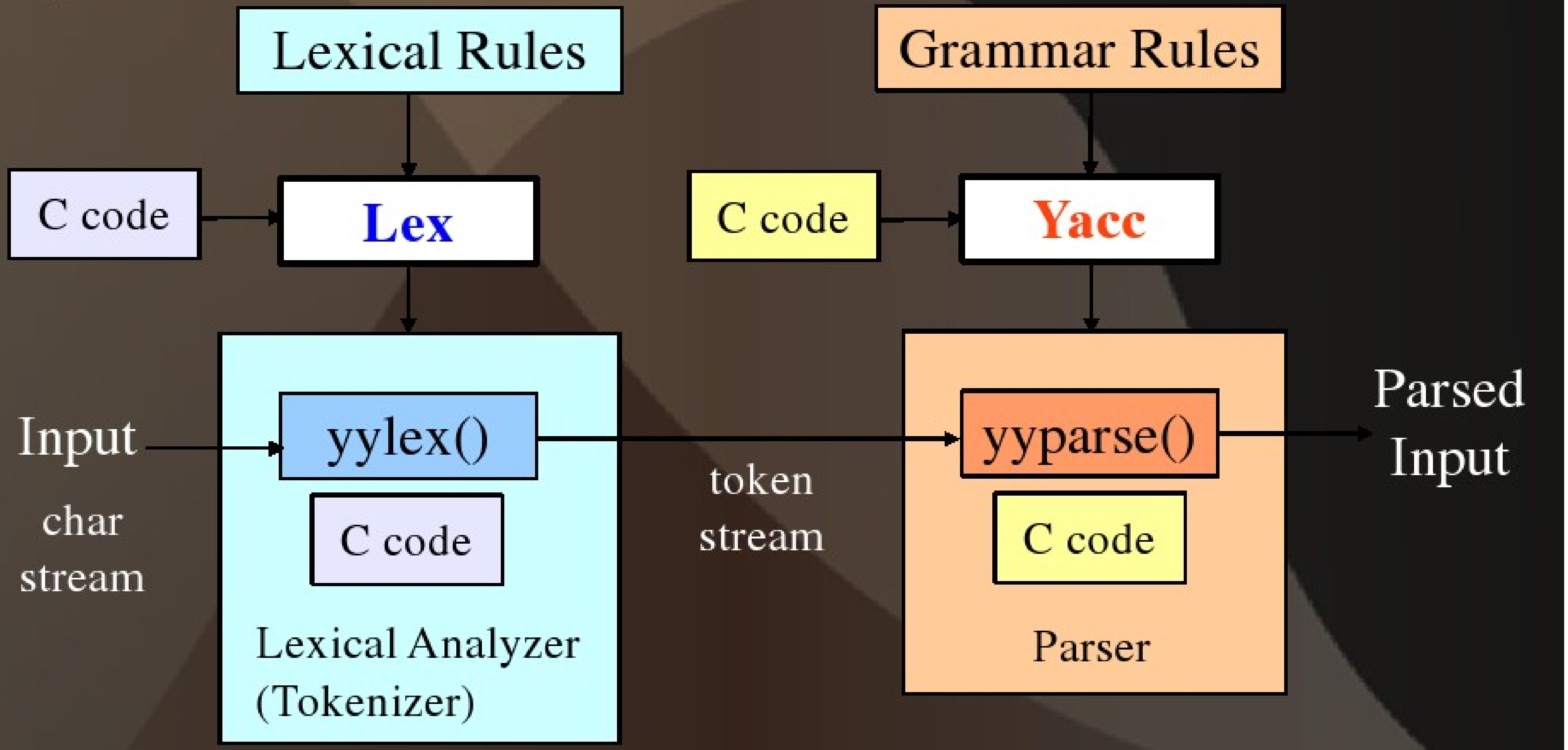
Lex and Yacc

- Two classical tools for compilers:
  - **Lex**: A Lexical Analyzer Generator
  - **Yacc**: "Yet Another Compiler Compiler" (Parser Generator)
- **Lex** creates programs that scan your tokens one by one.
- **Yacc** takes a grammar (sentence structure) and generates a parser.



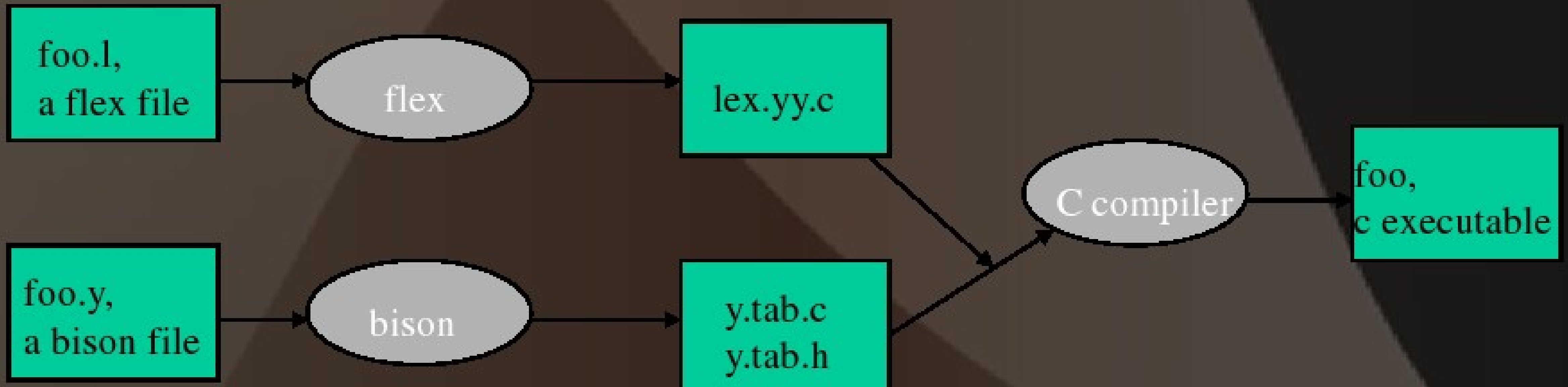
Lex and Yacc

- **Lex** and **Yacc** generate C code for your analyzer & parser.



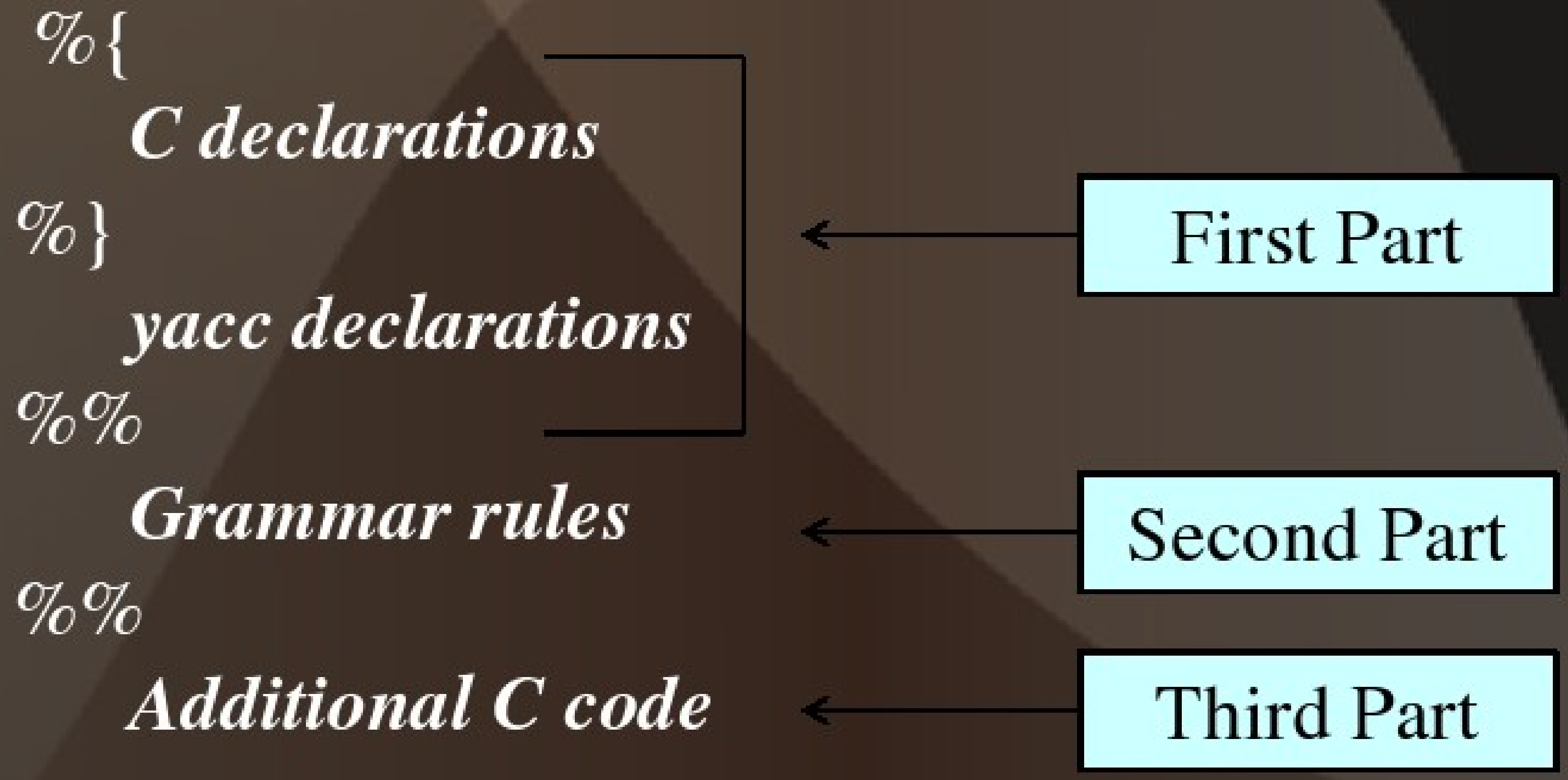
Flex, Bison

- Often, instead of the standard **Lex** and **Yacc**, **Flex** and **Bison** are used:
  - **Flex**: A fast lexical analyzer
  - (GNU) **Bison**: A replacement for (backwards compatible with) Yacc



Yacc: Input file format

- Yacc syntax is similar to Lex/Flex at the top level.
- Lex/Flex rules were "regular expression – action" pairs.
- Yacc rules are "grammar rule – action" pairs.



```
%{
    C declarations
%}
    yacc declarations
%%
    Grammar rules
%%
    Additional C code
```

Input File Format: First Part

- First Part includes:
  - C declarations enclosed in %{ %}
  - YACC definitions:
    - %**start**: Specify the grammar's start symbol
    - %**token**: Declare a terminal symbol (token type name) with no precedence or associativity specified
    - %**union**: Declare the collection of data types that semantic values may have
    - %**type**: Declare the type of semantic values for a non-terminal symbol
    - %**right**: Declare a terminal symbol (token type name) that is right-associative
    - %**left**: Declare a terminal symbol (token type name) that is left-associative

Yacc Productions: Second Part

- Represents the CFG, a set of productions
  - Format of production: **LHS : RHS**
  - Multiple RHS separated by ‘|’
  - Actions associated with a rule are entered within ‘{}’
- Example Productions:

```
statements : statement   { printf (“Statement”);}
           | statement statements
                         { printf (“Statements\n”);}
statement : id ‘+’ id    { printf (“plus\n”);}
           | id ‘-’ id   { printf (“minus\n”);}
```

Yacc Productions: Second Part

- **$1, $2, …, $n** refer to values associated with symbols on RHS
- **$$** refer to the value of the LHS
- Every symbol has a value associated with it (including tokens and non-terminals)
- Default action: $$ = $1
- Example Productions:

```
statement : id '+' id    { $$ = $1 + $2; }
          | id '-' id    { $$ = $1 - $2; }
```

- *When YACC processes these variables, it converts them into valid C for us.*

Auxiliary Procedures: Third Part

- Contains valid C code that supports language processing
  - Symbol table implementation
  - Functions that might be called by actions associated with the productions in the second part
    - First part may contain function prototypes with actual implementation in the third part

Yacc Example: Primitive Calculator

- int-valued calculator.
- Variable names are one character long; either a lower or upper case letter
- Example run:

```
$ ./calc
a = 1 +100;
print a;
Printing 101
B = a - 10;
print B;
Printing 91;
Print a + B;
Printing 192
Exit;
$
```

Yacc Example: Primitive Calculator

```
%{
  void yyerror (char *s);
  #include <stdio.h>
  #include <stdlib.h>
  int symbols [52];
  int symbolVal (char symbol);
  void updateSymbolVal (char
              symbol, int val);
%}
```

Called whenever there is a syntax error

A very primitive symbol table

Look-up symbol table for a value

Changes value of a symbol in the symbol table

Yacc Example: Primitive Calculator

```
%union { int num; char id;}
%start line
%token print
%token exit_command
%token <num> number
%token <id> identifier
%type <num> line exp type
%type <id> assignment
%%
```

Yacc Example: Primitive Calculator

```
/* Second Part */
line : assignment ‘;’   { ; }
   | exit_command ‘;’ { exit(EXIT_SUCCESS);}
   | print exp ‘;’    { printf(“Printing %d\n”, $2); }
   | line assignment ‘;’ { ; }
   | line print exp ‘;’ { printf(“Printing %d\n”, $3); }
   ;
assignment : identifier ‘=‘ exp {updateSymbolVal($1,$3);}
             ;
exp : term  {$$ = $1;}
   | exp ‘+’ term {$$ = $1 + $3;}
   | exp ‘-’ term {$$ = $1 - $3;}
   ;
term : number {$$ = $1;}
   | identifier {$$ = symbolVal ($1);}
%%
```

Yacc Example: Primitive Calculator

```
/* Third Part: C code */

int computeSymbolIndex (char token} {…}

int symbolVal (char symbol) {…}

void updateSymbolVal (char symbol, int val) {…}

int main () {…; return yyparse();}

void yyerror (char *s) {
  fprintf (stderr, %s\n”, s);
}
```

Yacc Example: The lex file

```
/* calc.l */
%{
  #include “y.tab.h”
%}

%%
“print”  {return print;}
“exit”   {return exit_command;}
[a-z][A-Z] {yylval.id = yytext[0]; return identifier;}
[0-9]+   {yylval.num = atoi(yytext); return number;}
[ \t\n]  ;
[-+=;]   {return yytext[0];}
.        {ECHO; yyerror(“Unexpected Char”);}

%%
int yywrap (void) {return 1;}
```

Yacc Example: Building

```
$ bison -vd calc.y
$ lex calc.l
$ gcc lex.yy.c y.tab.c -o calc
```

```
$ ./calc
a = 1 + 2;
print a;
Printing 3;
**
Unexpected Char
```