# Formal Languages and Compilers

## Lex and Yacc

# Lex and Yacc

- They are done to work in harmony
- Lex recognizes token
- Yacc builds the intermediate structure given the specifications
- Yacc calls iteratively Lex to parse the input
- We will build a simple calculator today by putting toghether lex and yacc

# Define the grammar

- We define the grammar we would like to parse, thus giving the structure to our input file.
- This is a simple calc, so we want to do arithmetic operations, eg:

  19 - 7 + 4

  (ideally something like E - E + E)
- Keep it simple for the while

# Define yacc specification

- What can we do with the calculator?
- What kind of production can we have?
- We need a grammar that is compliant with Yacc
- Any idea?
- S: …..

# Define yacc specification

- We need a way to define:
- NUMBERS
- SUM
- SUBTRACTION
- MULTIPLICATION
- DIVISION
- A starting symbol that reduces everything

# Define yacc specification

```
expr:
    ????
    | ???? '+' ????
    | ???? '-' ????
    ;
```

Starting symbol...

```
Program:        ????
                | ????
                ;
```

# Define yacc specification

```
expr:
    INTEGER
    | expr '+' expr
    | expr '-' expr
    ;
```

An expression is either a number or E + E or E – E.

program:

```
program:    expr '\n’
            |
            ;
```

# Define yacc specification

- Now that we have the grammar… what should we do?

```
expr:
        INTEGER              ????
        | expr '+' expr      ????
        | expr '-' expr      ????
        ;
```

# Define yacc specification

- We have to bind some actions to the production.
- Actions are executed when the production are **REDUCED**

```
expr:
    INTEGER                  ????
    | expr '+' expr          ????
    | expr '-' expr      ????
    ;
```

- We are working with a stack, Yacc take care of managing it for us with the use of $.. So…
  Let's see how do we bind items on the stack

# Define yacc specification

```
expr '+' expr        { $$ = $1 + $3; }
```

What does $$ identify?
Any guess?

# Define yacc specification

```
expr:
    INTEGER
    | expr '+' expr        { $$ = $1 + $3; }
    | expr '-' expr        { $$ = $1 - $3; }
    ;
```

An expression is either a number or E + E or E – E.

```
program:
    program expr '\n'      { printf("%d\n", $2); }
    |
  ;
```

# Define lexer rules

- Is time to build the Lexer
- So far we have just an INTEGER token, already specified in the Yacc file
- We will include y.tab.h in the lexer file, thus compiling at the first step the Yacc file
- We need a way to recognize integers and the symbol for plus and minus binary operators

# Lexer rules

```
[0-9]+      {
             yylval = atoi(yytext);
                        return INTEGER;
             }
[-+\n]      { return *yytext; }

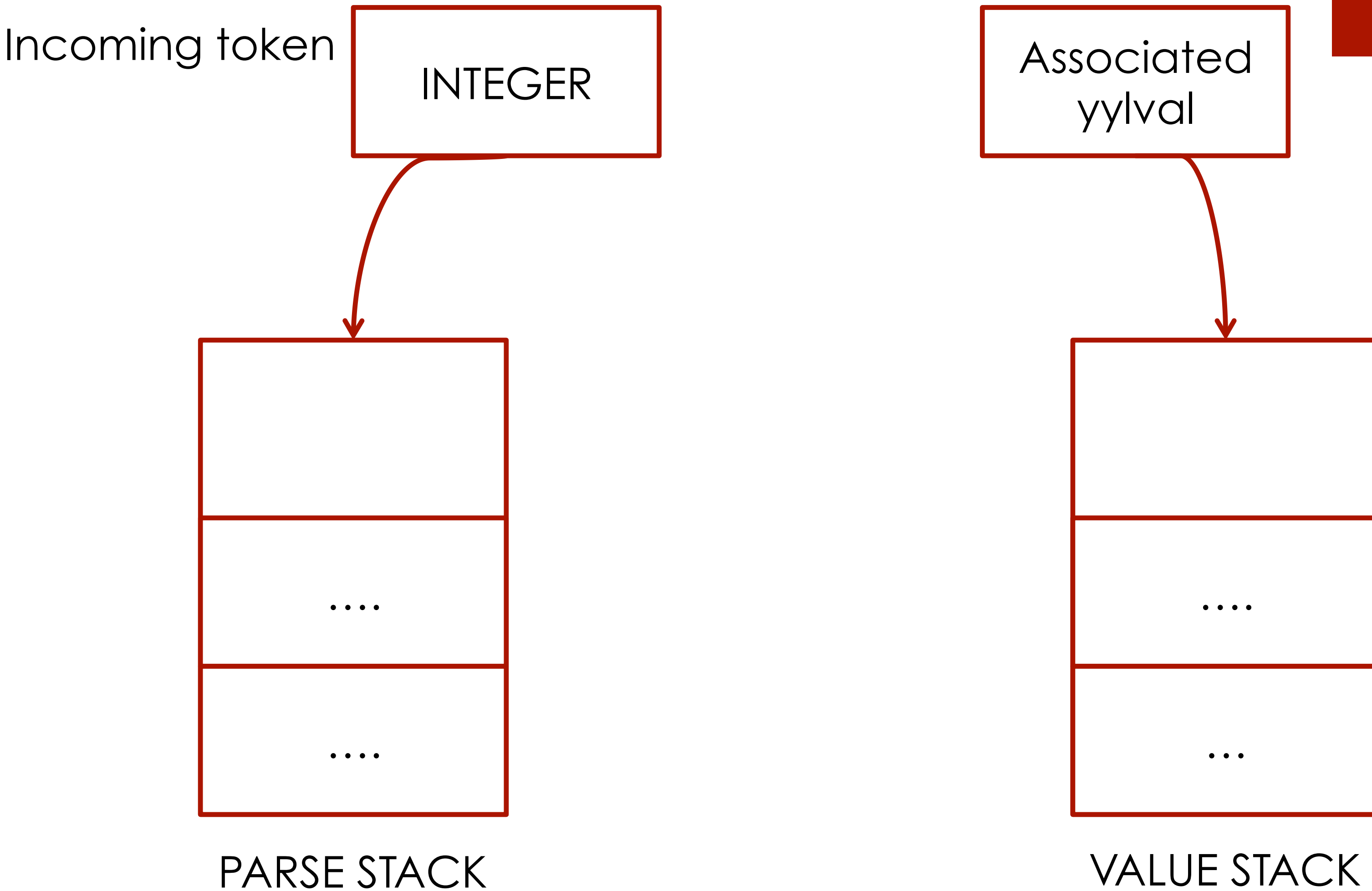
[ \t]       ;       /* skip whitespace */

.           yyerror("Unknown character");
```

# Yacc's double stack

- Yacc keeps track of two stacks
- Parse stack → terminals and non terminals
- Value stack → array of **YYSTYPE** elements
  **YYSTYPE …. typedef int YYSTYPE**; in y.tab.h..
- When Lex returns INTEGER as token → yacc undertakes some actions: it shift the token in the parse stack, and at the same moment the yylval corresponding is pushed into the value stack.

# Yacc's double stack



# Stack management example

- E : E1 + E2 { $$ = $1 +$3; }
  note: numbers are used to enumerate the production's left hand components
- You know that $1 represents E1, while $3 is the third term of the production, thus E2
- $$ is the top of the stack after the REDUCTION has taken place
- Action : sum the two values – pop off the values from the stack – pushes back the single sum
- Value and parse stack remain synchronized

# Action and stack

- What does happen when we have a terminal symbol…?
- E : INTEGER {…}
- Best guess?

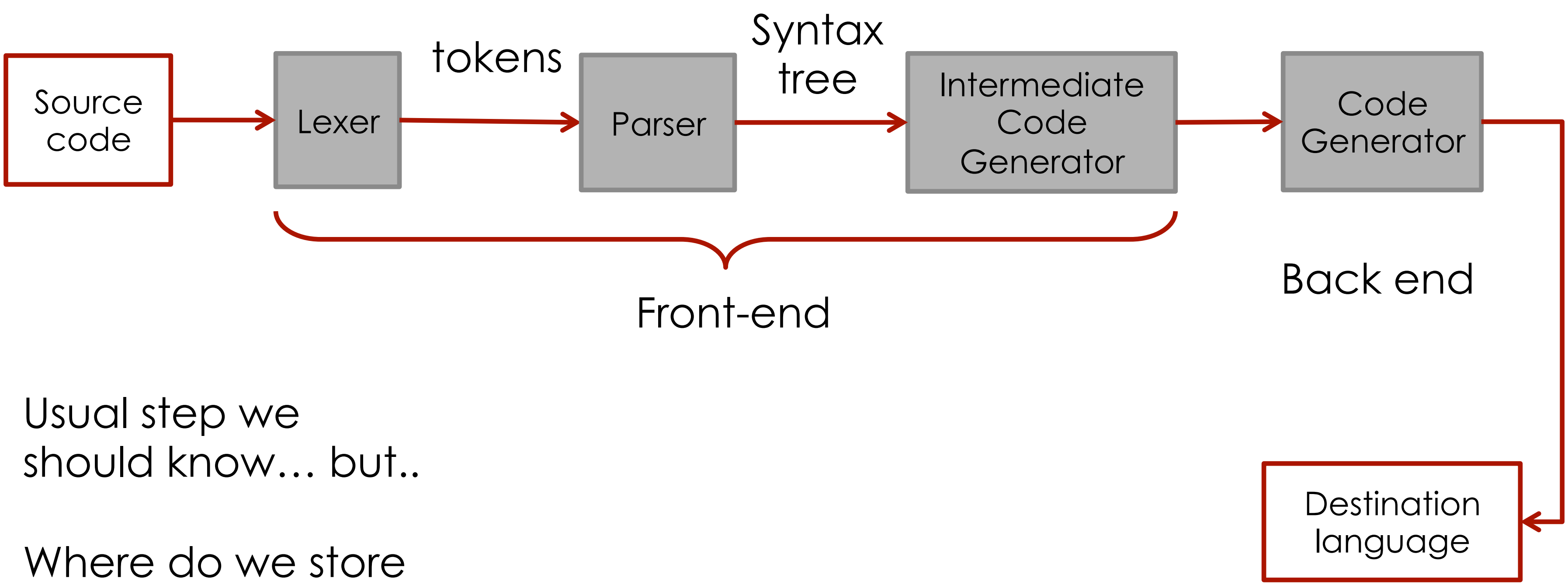
# Action and stack

- We just, simply, assign such value to the top of the stack
- E : INTEGER { $$ = $1;}
- This action is so common that is the default action. If we omit the action Yacc will try to do such assignment.
- Let's spend some time by looking at a complete example.
  directory 2.1 – calc lex and yacc

# Expand example above

- We want to use simple variable (say 1 char variable)
- We want to maintain variables values
- We want to add division, multiplication, parenthesis and assignation

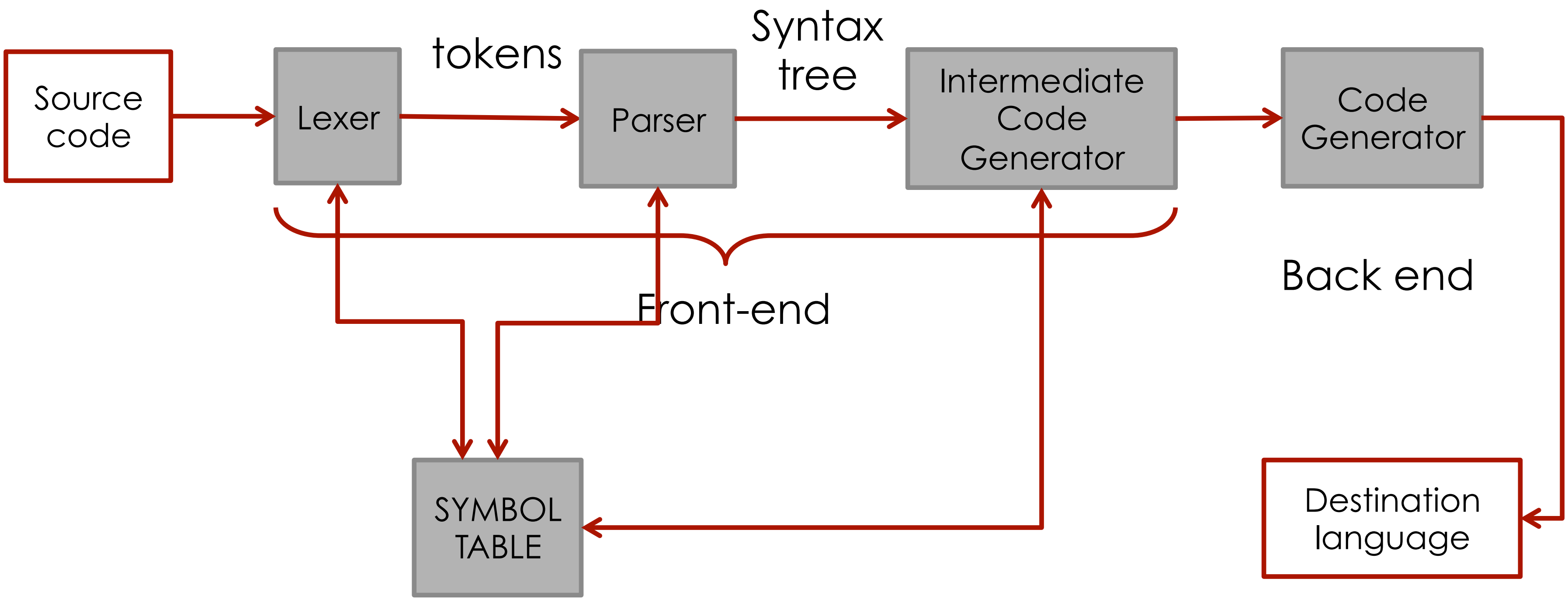
# Let's do a step back




Usual step we should know… but..

Where do we store variables???

# Let's do a step back



# Expand example above

- We want to use simple variable (say 1 char variable)
- We use a table to maintain the value of the variables
- We declare a ***symbol table*** (*int sym[26], we exploit C ansi char representation to store values)
- We want to add division, multiplication, parenthesis and value assignation

# Lexer and parser

- Lexer add on:
  A way to recognize single char variables
  What kind of regexp can we use?

# Lexer and parser

- Lexer add on:
  A way to recognize single char variables
- [a-z] will suffice
- We need some structure for the parser now, in order to assign variable's values
- We need to manage the new operations (/ * and ())
- Any guess how to do it?

# Lexer and parser

- Lexer add on:
  A way to recognize single char variables
- [a-z] will suffice
- We need structure for the parser now, to assign variables a value and to manage the new operations (/ * and ())
  S: E | var = E;
  E: … | variable | …. | E * E | E / E | ( E ) ;
  Do we need anything more?
- Example is in directory 2.1 – calc

# Interpreter vs Compiler

- An interpreter is able to execute the specified operation - in the program source file – directly onto the input data provided by the user
(eg. Perl v5 and minor)
- A compiler take the program source file and transforms it into an equivalent program written in another language (destination language)
(eg C, C++)
- Hybrid compilers: in the first step it compiles the source code to an intermediate code, such code is the interpreted together with the user input data (in a second step). (eg. Java – the bytecode and the virtual machine, some version of python)

# A full calculator example

- Let's introduce some flow control in our program, we start by adding construct such as if-then-else and while loop.
  EG:
  ```
  x = 0;
   while (x < 3) {
          print x;
          x = x + 1;
   }
  ```

- We will start by taking a look at the interpreter. The interpreted version of the above program produces the following output:
  **0**
  **1**
  **2**

# The aim

- We want to have some control over the flow of the program, and we want as well a way to parse complex operations
- (statements)
  S → ????
- (statements list)
  ….. → ….. | …..
- Any idea?

```
x = 0;
 while (x < 3) {
       print x;
       x = x + 1;
 }
```

# The aim

- We want to have some control over the flow of the program, and we want as well a way to parse complex operations

- The following is a draft of our grammar

- (statements)
  S → ';'
  | E ';' | print E ';' | var = E ';' | while (E) S
  | IF (E) S | IF (E) S ELSE S | S_list

- (S_list)
  S_list → S | S_list S

- (expressions)
  E : int | var | - E
  | E + E | E – E | E * E | E / E
  | E < E | E > E
  | E >= E | E <= E | E != E | E == E
  | ( E ) ;

- How to manage this situation? We have a E < E and E<=E… can Yacc manage this situation?
  Or E < E and E <= E ?
  Or E (var) = E and E == E ?
  Or –E and E-E
  What should we do?

- Starting the file structure:
  Program → ???

- (expressions)
  E : int | var | - E
  | E + E | E – E | E * E | E / E
  | E < E | E > E
  | E >= E | E <= E | E != E | E == E
  | ( E ) ;

- How to manage this situation? We have a E < E and E<=E… can Yacc manage this situation?
  Or E < E and E <= E ?
  Or E (var) = E and E == E ?
  Or –E and E-E
  What should we do?

- Starting the file structure:
  Program → function ;
  function → function statement | ε ;

# Interpreter – time to code

- We will divide our code in 4 files
- Calc.h → header file
- Calc.l → the lexer as usual
- Calc.y → the grammar
- calcXYZ.c → the evaluation procedure
  this will be our interpreter..

# Header for common decl.

- We need some data structure to hold some “things”…
- What kind of things?
- In our “concept” we have
- Constants → 1, 3, 35… (numbers)
- Variables → identifiers
- Operations

# Calc.h

```
typedef enum { typeCon, typeId, typeOpr } nodeEnum;

typedef struct {
        int value;
} conNodeType;

typedef struct {
        int I;
} idNodeType;

typedef struct {
        int oper;
        int nops;
        struct nodeTypeTag **op;
} oprNodeType;
```

Constant type

Identifiers type

Operation type

Let's define some useful structures in the header file. We will use such structures also for the compiler and the graph builder.

# Calc.h (cont.)

```
typedef struct nodeTypeTag {
   nodeEnum type;
   union {
      conNodeType con;
      idNodeType id;
      oprNodeType opr;
   };
} nodeType;
```

extern int sym[26]; → you know what that's for

You can find the whole file in the directory 2.2 - calc

# Calc.l – our lexer

- So far we defined the structures we need to manage the parsing action. We have still to define the lexer, the parser and the evaluation function.

- Lexer works at the same way already saw
  We need just some add-on to its rules.

0 if you find zero, just give back zero

[1-9][0-9]* any other number, return such number

NOTE: in the previous example (calc 2.1) integers with 2 numbers such as 11 or 89 would not be recognized.

# Calc.l (cont.)

- We need to recognize the token that act as *keywords* in our program such as ***while***, ***print*** or ***if*** ***else***….

```
">="        return GE;
"<="        return LE;
"=="        return EQ;
 "!="       return NE;
"while"     return WHILE;
"if"        return IF;
"else"      return ELSE;
"print"     return PRINT;
```

This does the trick for us, we recognize the token in the input using Lex and we give to Yacc a token to play with – which does not bring conflict in the parsing actions.

# Calc.y – the grammar

- This time we will use some user subroutines

```
nodeType *opr(int oper, int nops, ...);
nodeType *id(int i);
nodeType *con(int value);
void freeNode(nodeType *p);
int ex(nodeType *p);
int yylex(void);
void yyerror(char *s);
int sym[26];
```

- Just define them so that we can use such functions in the parsing action

# Calc.y (cont.)

- this cause a new type def to be generated,which is a union of the above and is calledYYSTYPE. Then there is the declaration extern YYSTYPE yylval which declares yylval as an external variable

- ```
  %union {
      int iValue;          /* integer value */
      char sIndex;         /* symbol table index */
      nodeType *nPtr;      /* node pointer */
  };
  ```

- Take a look at y.tab.h and look at what you find there
  token type + our newly defined union of type YYSTYPE
  ```

# Advanced yylval

- Integer iValue
- Char sIndex
- nodeType pointer *nPtr
- Extended token syntax
  %token <iValue> INTEGER

  %token <sIndex> VARIABLE
- When reference with $ is used in the action, Yacc *auto magically* access the right member of the union for us. (he just takes care for us to do the job) thanks to the previous specification

# Subroutines

- The rest of the file contains the grammar and the user routines.
- We will proceed by analysing them..
- nodeType *id(int i){…} → returns a node that contains the identifier name
- nodeType *con(int value) {…}→ return a node that holds the constant value – when we find an integer

# Subroutines (cont.)

nodeType *opr(int oper, int nops, ...){…}

this function builds and returns a node that represents the operation that we are tackling during the parsing action.
We save information about the operation and its 'argument'

# The interpreter

- Finally last step.
  The 4th file will do the job for us.
- We need the clac.h and y.tab.h → token and structures definitions
- One function called ex(ecute) – you can call it eval if it sounds more familiar with your previous knowledge
- What should this function do?

# The interpreter (cont.)

Take the node p, which is the top of our tree so far..

IF is a constant → return constant value
IF is a identifier → return the value in the symbol table
IF is an operation → switch on the operation type
    case WHILE execute the while by evaluating the guard
    case IF check if has and else and its guard
    case PRINT → print the expression..

Let's take a look at the file and let's understand how does it work,
File is calcInterpreter.c in directory 2.2 - calc

# Bibliography


- Tom Niemann – Lex and Yacc tutorial
  epaperpress.com/lexandyacc
- Compilers 2nd edition – Aho, Lam, Sethi, Ullman
- The linux documentation project
  http://tldp.org/HOWTO/Lex-YACC-HOWTO-6.html