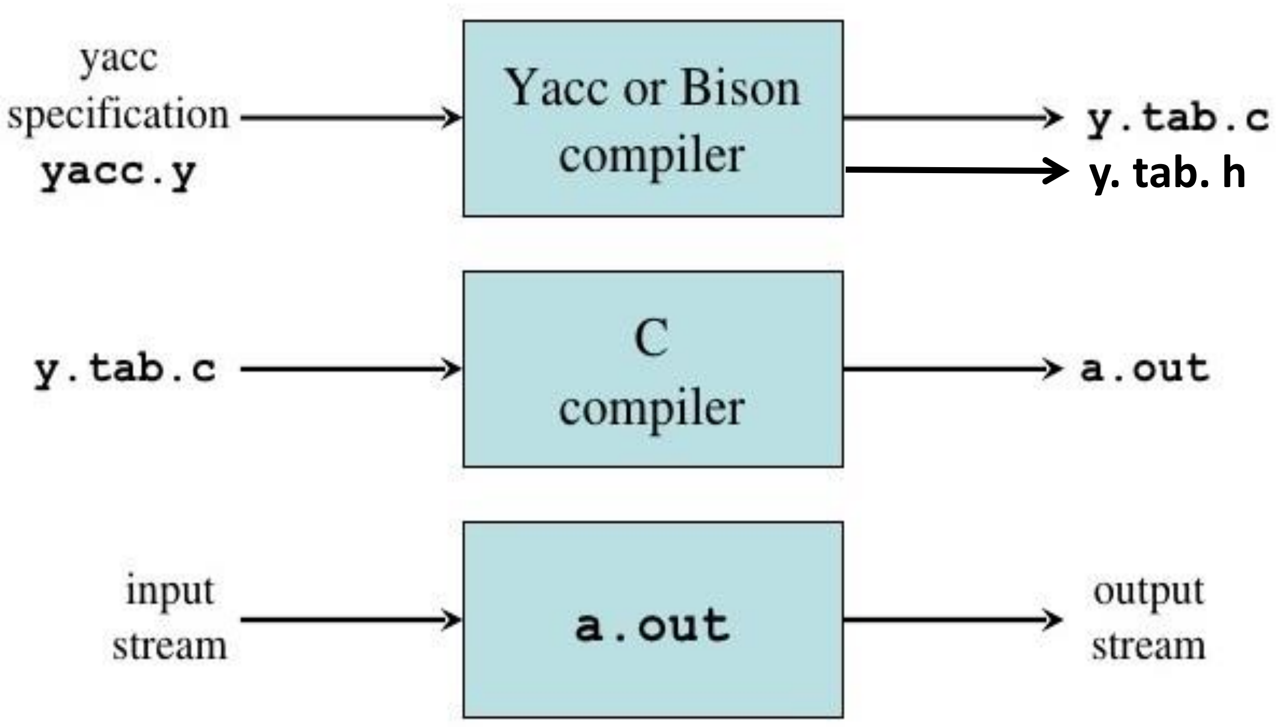
Bison/ YACC

Yacc

- **Yacc** stands for yet another compiler compiler.
- Designed to compile grammar given as input.
- Generate **LALR** parser to recognize sentences in the grammar.
- **Bison** is the updated version of **Yacc**.

Creating LALR parser with YACC



How Yacc Works (Cont…)

- Designed for use with **C** code.
- Generates a parser written in **C**.
- Parser configured for use in conjunction with a **lex**- generated scanner.
- Relies on shared features (token types, **yylval**, etc.).
- Provide grammar specification file with **.y** extension.

Shift and Reducing

- Perform Shift/reduce parsing
- Maintains set of states, reflecting one or more partially parsed rules
- After reading a token it may take two possible actions
  - **Shift:** If the token cannot complete any rule, shift the Token from input in internal stack
  - **Reduce**: If a rule can be completed, then pop all R.H.S. symbol from the stack and push L.H.S. symbol

Shift and Reduce Parsing

E-> E+E | E-E | E*E | E/E | (E) | NUM

Stack	Input	Action
$	(num+num)*num $	Shift (
$(	num+num)*num $	Shift num
$(num	+num)*num $	Reduce E->num
$(E	+num)*num $	Shift +
$(E+	num)*num $	Shift num
$(E+ num	)*num $	Reduce E->num
$(E+E	)*num $	Reduce E->E+E
$(E	)*num $	Shift)
$(E)	*num $	Reduce E-> (E)
$E	*num $	Shift *
$E*	num $	Shift num
$E* num	$	Reduce E->num
$E*E	$	Reduce E->E*E
$E	$	Accept

Scanner Parser Interaction

- Parser assumes the existence of yylex() function
- This is called by yyparse()
- How do both scanner and parser agree on the TOKEN names??
  - Define token names in YACC program
  - After compiling “bison –d”, a y.tab.h file in produced
  - This contains the enumeration of token definition
  - So, need to include the “y.tab.h” file in lex source code

Scanner Parser Interaction

- With each token scanner can send some value associated with it using global variable yylval
- Default type of yylval is int
- We can redefine it’s type
  - Will see an example later

How Yacc Works (Cont…)

- Invoke **Yacc** on the **.y** file by calling **yyparse()** function.
- Creates the **y.tab.h** and **y.tab.c** files.
- File provides an extern function **yyparse().**
- **yyparse()** returns a value of 0 if the input it parses is valid according to the given grammar rules. It returns a 1 if the input is incorrect and error recovery is impossible, gives syntax error.
- It calls a routine called **yylex()** every time it wants to obtain a token from the input.
- **yylex()** returns a value indicating the *type* of token that has been obtained.

Yacc File Format

- Input to Yacc is divided into three sections.

```
... definitions ...
%%
... production rules ...
%%
... subroutines...
```

Yacc File Format (Cont…)

- **The definitions section consists of:**
  - token declarations .
  - C code bracketed by “%{“ and “%}”.

- **The rules section consists of:**
  - grammar

- **The subroutines section consists of:**
  - user subroutines .
  - Usually contains user defined main function

The Grammar

E-> E+T | T
T-> T*F | F
F-> digit | (E)

Here,
Non-terminals: E, T, F
Terminals/ Tokens: +, *, (,),
digit

```
expr : expr '+' term
| term
;
term : term '*' factor
| factor;

factor : DIGIT
| '(' expr ')'
;
```

Convention:
Non-terminals: lower-case
Terminal: upper-case, symbol enclosed in ‘ ’

The Grammar (Cont…)

- **expr, term and factor are nonterminals.**
- **DIGIT, '+', '*', '(', ')' are terminals** (tokens returned by lex).
- **Expression may be :**
  - sum of two expressions .
  - product of two expressions .
  - or a digit

Yacc token Declaration

- **‘%start’:** Specifies the grammar’s start symbol.
- **‘%token’:** Declare a terminal symbol with no precedence or associativity specified.
- **‘%left’:** Declare a terminal symbol that is left-associative .
- **‘%right’:** Declare a terminal symbol that is right- associative .
- **%nonassoc** Identifies tokens that are not associative with other tokens.
- %**type:** Identifies the type of nonterminals. Type-checking is performed when this construct is present.
- %**union** Identifies the yacc value stack as the union of the various type of values desired. By default, the values returned are integers. The effect of this construct is to provide the declaration of **YYSTYPE** directly from the input.

Token

- In yacc file token definition:

```
%token NUM
```

- In y.tab.h file:

```
#define NUM 258
```

- The lex file can return NUM
- Definitions usually starts from 258 in y.tab.h

Production rules

```
expr : expr ‘+’ term                    {$$=$1 + $3;}
| term                                  {$$=$1; //by default}
;
term : term ‘*’ factor                  {$$=$1 * $3;}
| factor;

factor : DIGIT
| ‘(‘ expr ‘)’                          {$$=$2;}
;
```

Production rules

```
expr : expr '+' term                    {$$=$1 + $3;}
| term                                  {$$=$1; //by default}
;
term : term '*' factor                  {$$=$1 * $3;}
| factor;

factor : DIGIT
| '(' expr ')'                          {$$=$2;}
;
```

- Note that the nonterminal `term` in the first production is the third grammar symbol of the body, while `+` is the second
- The semantic action associated with the first production adds the value of the `expr` and the `term` of the body and assigns the result as the value for the nonterminal `expr` of the head.

Production rules

```
expr : expr ‘+’ term            {$$=$1 + $3;}
| term                          {$$=$1; //by default}
;
term : term ‘*’ factor          {$$=$1 * $3;}
| factor;

factor : DIGIT
| ‘(‘ expr ‘)’                  {$$=$2;}
;
```

- In a semantic action, the symbol `$$` refers to the attribute value associated with the nonterminal of the head.
- While `$i` refers to the value associated with the *i*th grammar symbol (terminal or nonterminal) of the body.

Production rules

```
 $$      $1  $2  $3
 ↓       ↓   ↓   ↓
expr : expr '+' term                {$$=$1 + $3;}
| term                              {$$=$1; //by default}
;
term : term '*' factor              {$$=$1 * $3;}
| factor;

factor : DIGIT
| '(' expr ')'                      {$$=$2;}
;
```

- In a semantic action, the symbol `$$` refers to the attribute value associated with the nonterminal of the head.
- While `$i` refers to the value associated with the *i*th grammar symbol (terminal or nonterminal) of the body.

yyparse()

- Called once from main()
- Repeatedly calls yylex()
- On syntax error, calls yyerror(char *s)
- Returns 0 if all of the input is processed
- Returns 1 if aborting due to syntax error

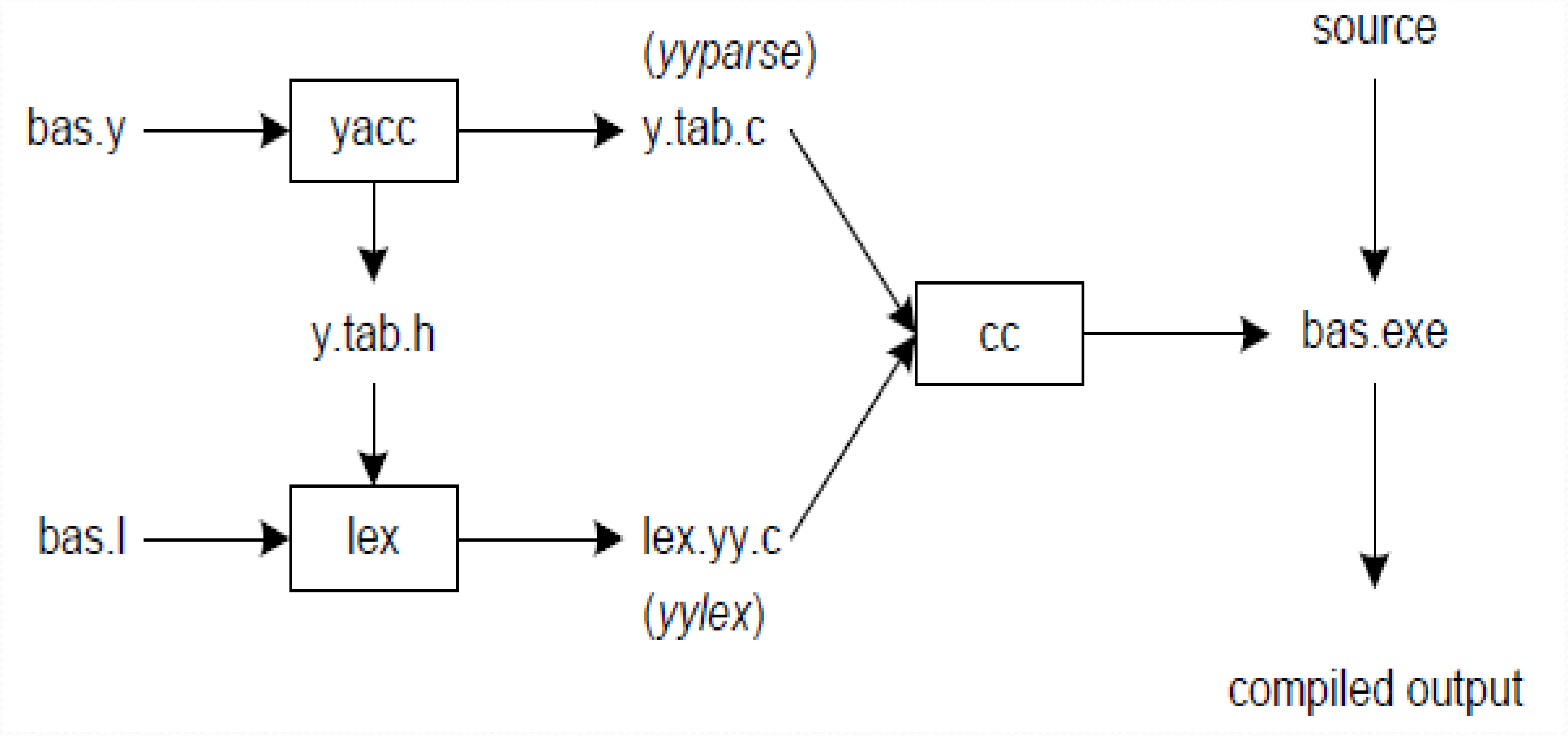
yyerror()

- The function that the parser calls upon encountering an input error. The default function, defined in liby.a, simply prints string to the standard error. The user can redefine the function. The function's type is void.

error token

- The specially defined token error matches any unrecognized sequence of input.
- This token causes the parser to invoke the yyerror function.
- By default, the parser tries to synchronize with the input and continue processing it by reading and discarding all input up to the symbol following error.
- If no error token appears in the yacc input file, the parser exits with an error message upon encountering unrecognized input.

Linking Lex & Yacc



Command

- Compile Lex File to generate lex.yy.c file:

```
flex simplecalc.l
```

- Compile YACC File to generate y.tab.c and y.tab.h file:

```
bison -d simplecalc.y //will produce simplecalc.tab.c and simplecalc.tab.h
or,
bison –d -y simplecalc.y //will produce y.tab.c and y.tab.h
```

- Use g++ Compiler to link lex.yy.c and y.tab.c files:

```
g++ lex.yy.c  simplecalc.tab.c
```

Some Resourceful Link

- Operator Precedence Link:

http://en.cppreference.com/w/c/language/operator_precedence

- Bison Software Download Link:

http://gnuwin32.sourceforge.net/packages/bison.htm

Thank You