

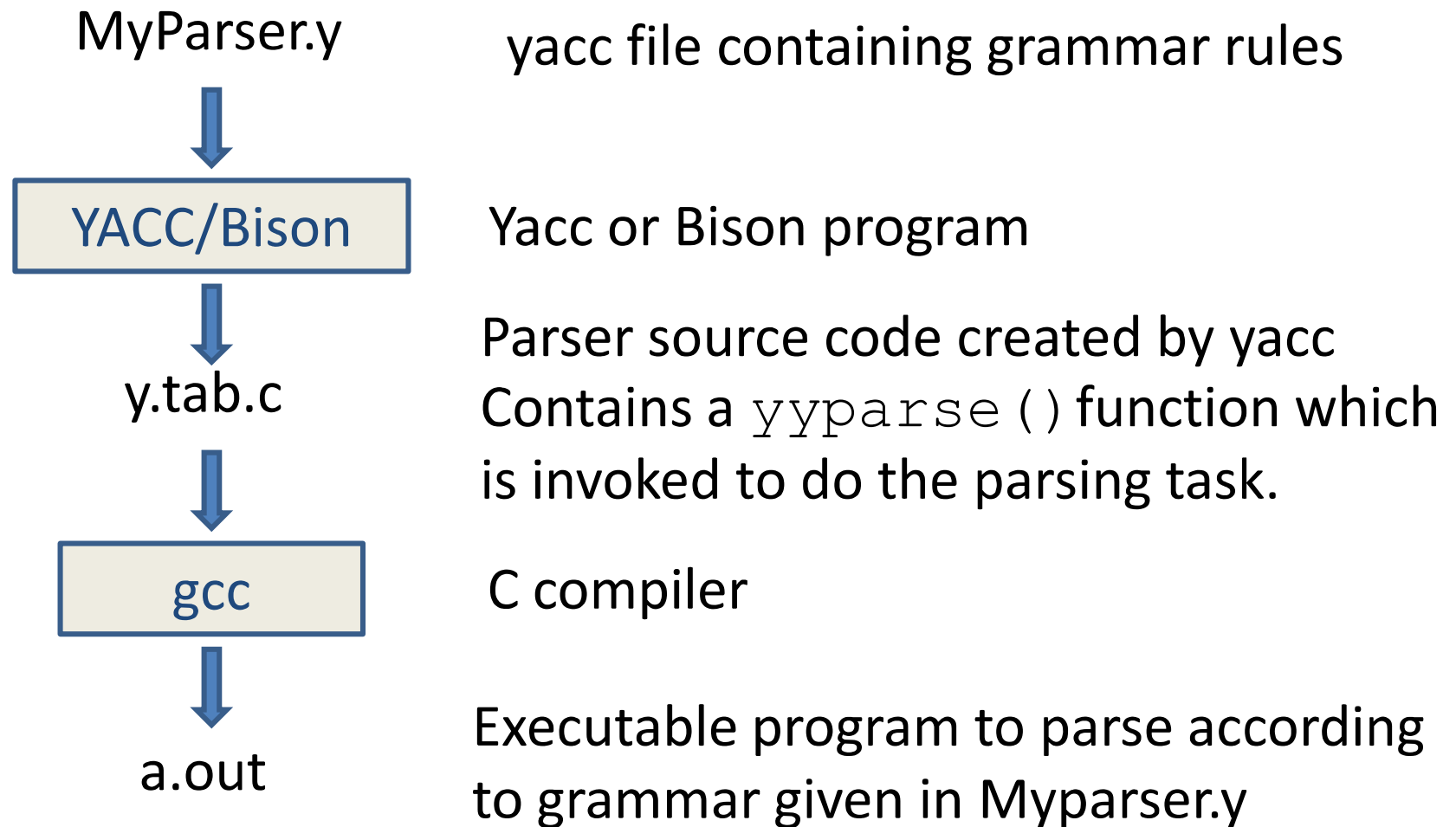
CSE 310

YACC (or Bison)

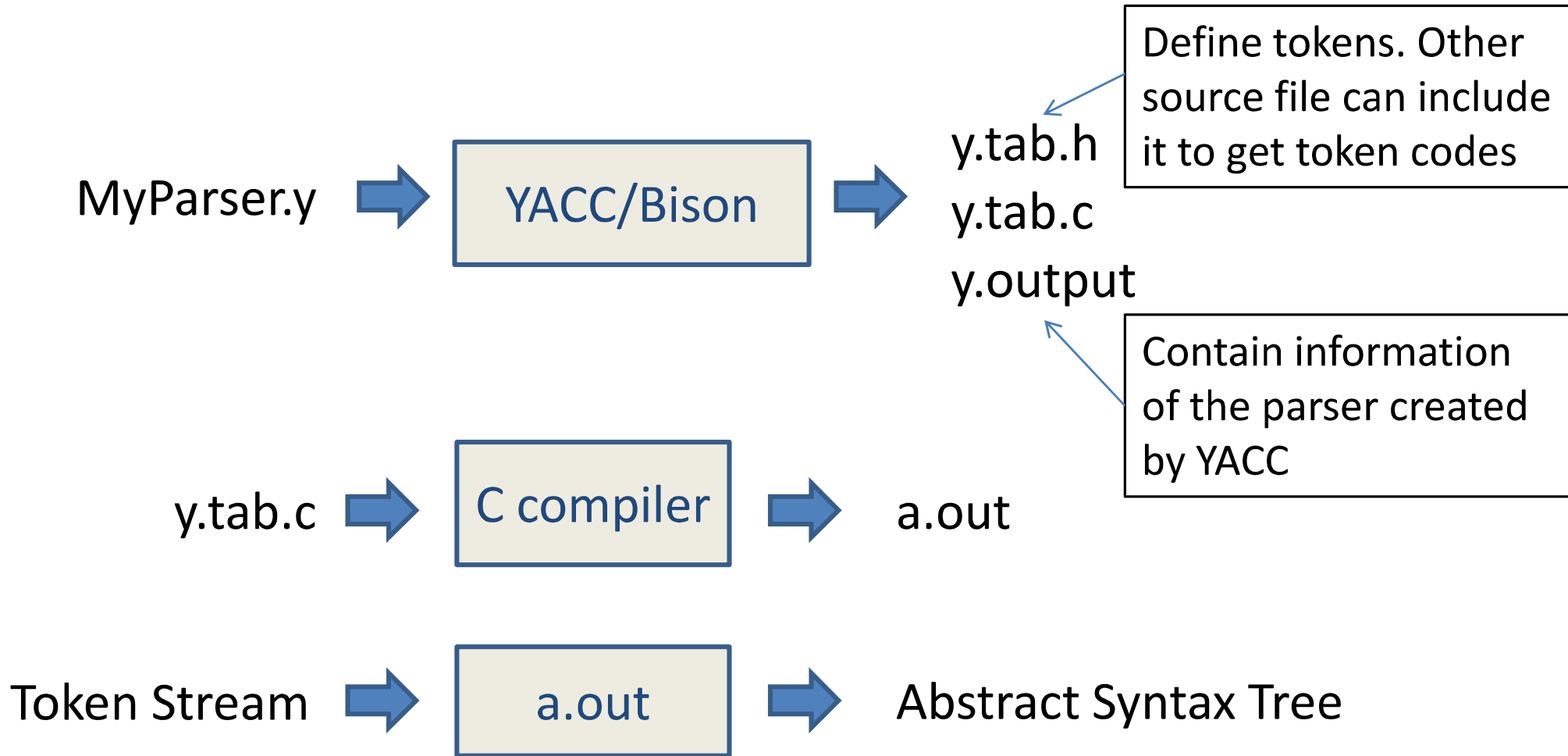
YACC, Bison

- **Yet Another Compilers Compiler**
- Unix utility that parses a token stream produced by lex according to specified LALR(1) context free grammar
- We will use **Bison** which is YACC compatible

How to use YACC



How to use YACC



Basics: How Bison Works?

- Perform Shift/reduce parsing
 - So that's bottom-up 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 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 reducing

```
stmt: stmt ';' stmt
      | NAME '=' exp
```

```
exp: exp '+' exp
     | exp '-' exp
     | NUMBER
```

```
stack:
<empty>
```

```
input:
a = 7; b = 3 + a + 2
```

Shift and reducing

```
stmt: stmt ';' stmt
      | NAME '=' exp
```

```
exp: exp '+' exp
     | exp '-' exp
     | NUMBER
```

SHIFT!

stack:

NAME

input:

= 7; b = 3 + a + 2

Shift and reducing

```
stmt: stmt ';' stmt
      | NAME '=' exp
```

```
exp: exp '+' exp
     | exp '-' exp
     | NUMBER
```

SHIFT!

stack:
NAME '='

input:
7; b = 3 + a + 2

Shift and reducing

```
stmt: stmt ';' stmt
      | NAME '=' exp
```

```
exp: exp '+' exp
     | exp '-' exp
     | NUMBER
```

SHIFT!

```
stack:
NAME '=' 7
```

```
input:
; b = 3 + a + 2
```

Shift and reducing

```
stmt: stmt ';' stmt
      | NAME '=' exp
```

```
exp: exp '+' exp
     | exp '-' exp
     | NUMBER
```

REDUCE!

```
stack:
NAME '=' exp
```

```
input:
; b = 3 + a + 2
```

Shift and reducing

```
stmt: stmt ';' stmt
      | NAME '=' exp
```

```
exp: exp '+' exp
     | exp '-' exp
     | NUMBER
```

REDUCE!

stack:
stmt

input:
; b = 3 + a + 2

Shift and reducing

```
stmt: stmt ';' stmt
      | NAME '=' exp
```

```
exp: exp '+' exp
     | exp '-' exp
     | NUMBER
```

SHIFT!

```
stack:
stmt ';'

```

```
input:
b = 3 + a + 2

```

Shift and reducing

```
stmt: stmt ';' stmt  
      | NAME '=' exp
```

```
exp: exp '+' exp  
     | exp '-' exp  
     | NUMBER
```

SHIFT!

```
stack:  
stmt ';' NAME
```

```
input:  
= 3 + a + 2
```

Shift and reducing

```
stmt: stmt ';' stmt  
      | NAME '=' exp
```

```
exp: exp '+' exp  
     | exp '-' exp  
     | NUMBER
```

SHIFT!

```
stack:  
stmt ';' NAME '='
```

```
input:  
3 + a + 2
```

Shift and reducing

```
stmt: stmt ';' stmt
      | NAME '=' exp
```

```
exp: exp '+' exp
     | exp '-' exp
     | NUMBER
```

SHIFT!

```
stack:
stmt ';' NAME '=' NUMBER
```

```
input:
+ a + 2
```

Shift and reducing

```
stmt: stmt ';' stmt
      | NAME '=' exp
```

```
exp: exp '+' exp
     | exp '-' exp
     | NUMBER
```

REDUCE!

```
stack:
stmt ';' NAME '=' exp
```

```
input:
+ a + 2
```


Shift and reducing

```
stmt: stmt ';' stmt
      | NAME '=' exp
```

```
exp: exp '+' exp
     | exp '-' exp
     | NUMBER
```

SHIFT!

```
stack:
stmt ';' NAME '=' exp '+'
```

```
input:
a + 2
```

Shift and reducing

```
stmt: stmt ';' stmt
      | NAME '=' exp
```

```
exp: exp '+' exp
     | exp '-' exp
     | NUMBER
```

SHIFT!

```
stack:
stmt ';' NAME '=' exp '+'
NAME
```

```
input:
+ 2
```

Shift and reducing

```
stmt: stmt ';' stmt
      | NAME '=' exp
```

```
exp: exp '+' exp
     | exp '-' exp
     | NUMBER
```

REDUCE!

```
stack:
stmt ';' NAME '=' exp '+'
exp
```

```
input:
+ 2
```

Shift and reducing

```
stmt: stmt ';' stmt
      | NAME '=' exp
```

```
exp: exp '+' exp
     | exp '-' exp
     | NUMBER
```

REDUCE!

```
stack:
stmt ';' NAME '=' exp
```

```
input:
+ 2
```

Shift and reducing

```
stmt: stmt ';' stmt
      | NAME '=' exp
```

```
exp: exp '+' exp
     | exp '-' exp
     | NUMBER
```

SHIFT!

```
stack:
stmt ';' NAME '=' exp '+'
```

```
input:
2
```

Shift and reducing

```
stmt: stmt ';' stmt
      | NAME '=' exp
```

```
exp: exp '+' exp
     | exp '-' exp
     | NUMBER
```

SHIFT!

```
stack:
stmt ';' NAME '=' exp '+'
NUMBER
```

```
input:
<empty>
```

Shift and reducing

```
stmt: stmt ';' stmt
      | NAME '=' exp
```

```
exp: exp '+' exp
     | exp '-' exp
     | NUMBER
```

REDUCE!

```
stack:
stmt ';' NAME '=' exp '+'
exp
```

```
input:
<empty>
```

Shift and reducing

```
stmt: stmt ';' stmt
      | NAME '=' exp
```

```
exp: exp '+' exp
     | exp '-' exp
     | NUMBER
```

REDUCE!

```
stack:
stmt ';' NAME '=' exp
```

```
input:
<empty>
```


Shift and reducing

```
stmt: stmt ';' stmt
      | NAME '=' exp
```

```
exp: exp '+' exp
     | exp '-' exp
     | NUMBER
```

REDUCE!

```
stack:
stmt ';' stmt
```

```
input:
<empty>
```

Shift and reducing

```
stmt: stmt ';' stmt
      | NAME '=' exp
```

```
exp: exp '+' exp
     | exp '-' exp
     | NUMBER
```

REDUCE!

stack:
stmt

input:
<empty>

Shift and reducing

```
stmt: stmt ';' stmt  
      | NAME '=' exp
```

```
exp: exp '+' exp  
     | exp '-' exp  
     | NUMBER
```

DONE!

stack:
stmt

input:
<empty>

How parser get Tokens?

- We constructed lexical analyzer that generate tokens!
- So somehow our parser should communicate with scanner!

Scanner Parser Interaction

- Parser assumes the existence of `yylex()` function
- This is called by `yyparse()`
- But how they agree on Token names?

Scanner Parser Interaction

- Parser assumes the existence of `yylex()` function
- This is called by `yyparse()`
- But how they agree on Token names?
 - Define token names in YACC program.
 - If you compile using “`yacc -d`” or “`bison -d`”, it will produce a `y.tab.h` file
 - This contains an enumeration of token definition
 - Include this `y.tab.h` file in your lex source file

Scanner Parser Interaction

- With each token scanner can send some value associated with it using global variable `yyval`.
- Default type is `int`
- You can redefine its type
 - We will see example
- `yytext` is also available to your parser

YACC file format

- An yacc file has **.y** extension
- Three Sections.

```
/**** Definition Section *****/  
%%  
/**** Rules Section *****/  
%%  
/**** User SubRoutines *****/
```

- Looks Familiar??

Definition Section

- Any code within %{ and %} is copied in the output file of yacc program.
- Name of the tokens (%token)
- Associativity and precedence rules (%left, %right, %nonassoc)
- Name of the start symbol (%start)
- Data type of value associated with each token (%union)
- Several other things

Definition Section

```
% {
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
% }
```

```
% token ID NUM
```



Terminal

```
% type variable
```

Non
Terminal

```
% start expr
```

Rules Section

- Grammar rules and corresponding actions in C code.

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

term  : term '*' factor    { $$ = $1 * $3; }
      | factor             { $$ = $1; }
      ;

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

User Subroutine Section

- C codes copied directly in the `y.tab.c` file
- Usually contains user defined main function
- Call **`yyparse()`** from main function

Token

- In yacc file token definitions:

```
%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

yyval

- Global variable that can be used to return some values along with token
- Declare it in lex file
- Data type is `YYSTYPE` which is defined as `int` by default
- Lex Program:

```
extern int yyval;  
%%  
[0-9]+      { yyval= atoi(yytext);  
              return NUM;  
            }
```

yyval

- If different token requires different values, you can define a union in yacc file and associate appropriate values for a Token
- Use `union`
- See an example!

Precedence & Associativity

%left '+' , '-'

%left '*' , '/'

%nonassoc UMINUS

%%

expr	:	expr '+' expr	{ \$\$=\$1+\$2; }
		expr '-' expr	{ \$\$=\$1-\$2; }
		expr '*' expr	{ \$\$=\$1*\$2; }
		expr '/' expr	{ \$\$=\$1/\$2; }
		'-' expr %prec UMINUS	{ \$\$=-\$2; }
		NUM	{ }

Accessing Value Stack

- We can use \$ to access value returned by lex

```
expr : expr '+' expr    { $$=$1+$2; }  
      | expr '-' expr    { $$=$1-$2; }  
      | expr '*' expr    { $$=$1*$2; }  
      | expr '/' expr    { $$=$1/$2; }  
      | '-' expr         { $$=-$2; }  
      | NUM              {   }
```

Recursive Grammar

- Left recursion

```
list:  
    item  
    | list ',' item  
    ;
```

- Right recursion

```
list:  
    item  
    | item ',' list  
    ;
```

- YACC, Bison prefers left recursion. Why?

Conflicts

- Conflicts arise when there is more than one way to proceed with parsing.
- Two types:
 - shift-reduce [default action: *shift*]
 - reduce-reduce [default: *reduce with the first rule listed*]

Conflicts

- Reduce/Reduce Conflict:

A : B | C

B : 'X'

C : 'X'

- Shift/Reduce Conflict:

Stmt: IF '(' exp ')' stmt

 | IF '(' exp ')' stmt else stmt

Handling Conflicts

- See the `y.output` file for conflict details
- Think out why conflict occurred
- Rewrite grammar accordingly

```
int yyparse()
```

- Called once from `main()`
- Repeatedly calls `yylex()` until done:
 - On syntax error, calls `yyerror(char *s)`
 - Returns 0 if all of the input was processed;
 - Returns 1 if aborting due to syntax error.

Error Reporting

- Define the `yyerror(char *)` function
- Setting ‘%error-verbose’ in definition section can produce more information about error

Error Handling

- The “token” ‘**error**’ is reserved for error handling:
 - can be used in rules;
 - suggests places where errors might be detected and recovery can occur.

Example:

```
stmt : IF '(' expr ')' stmt
      | IF '(' error ')' stmt
      | FOR ...
      | ...
```


Error Handling

When an error occurs, the parser:

- pops its stack until it enters a state where the token ‘error’ is legal;
- then behaves as if it saw the token ‘error’
 - performs the action encountered;
 - resets the lookahead token to the token that caused the error.
- If no ‘error’ rules specified, processing halts.

YACC Declaration Summary

`%start'

Specify the grammar's start symbol

`%union'

Declare the collection of data types that semantic values may have

`%token'

Declare a terminal symbol (token type name) with no precedence or associativity specified

`%type'

Declare the type of semantic values for a nonterminal symbol

YACC Declaration Summary

`%right'

Declare a terminal symbol (token type name) that is right-associative

`%left'

Declare a terminal symbol (token type name) that is left-associative

`%nonassoc'

Declare a terminal symbol (token type name) that is nonassociative (using it in a way that would be associative is a syntax error, ex: $x \text{ op. } y \text{ op. } z$ is syntax error)

Reference

- Flex & Bison by John Levine
- Lecture on YACC by Tanvir Ahmed Khan
- Powerpoint slide from
www.csie.ntu.edu.tw/~b92006/YACC-present-2005.ppt
- Powerpoint slide from
www.cs.arizona.edu/~debray/Teaching/CSc453/DOCS/yacc%20tutorial.ppt

Thank You!