YACC

YACC

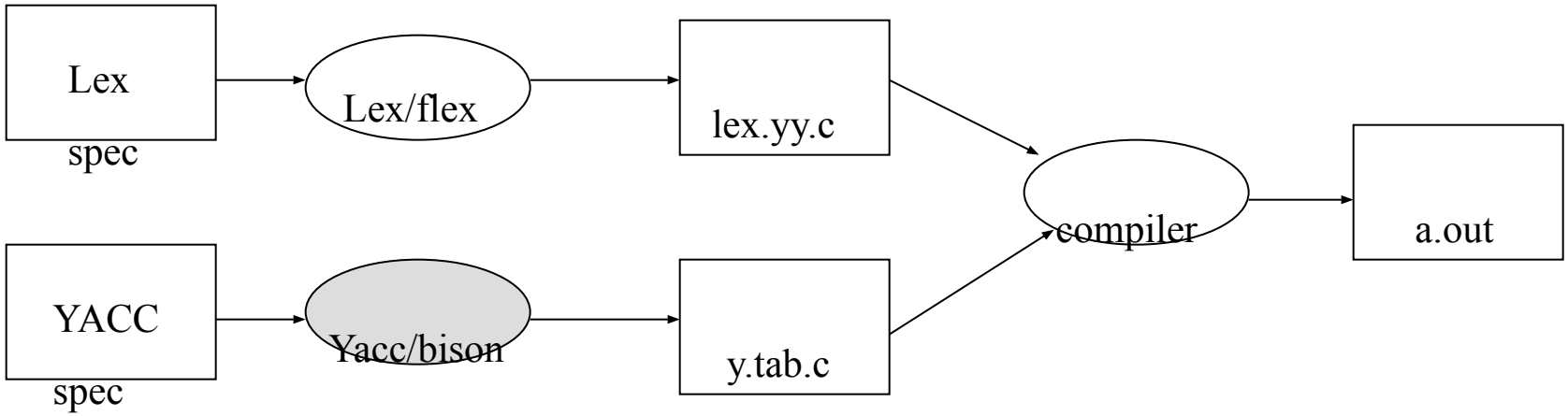
- **Tool which will produce a parser for a given grammar.**
- YACC (Yet Another Compiler Compiler) is a program designed to compile a LALR(1) grammar and to produce the source code of the syntactic analyzer of the language produced by this grammar
- Input is a grammar (rules) and actions to take upon recognizing a rule
- Output is a C program and optionally a header file of tokens

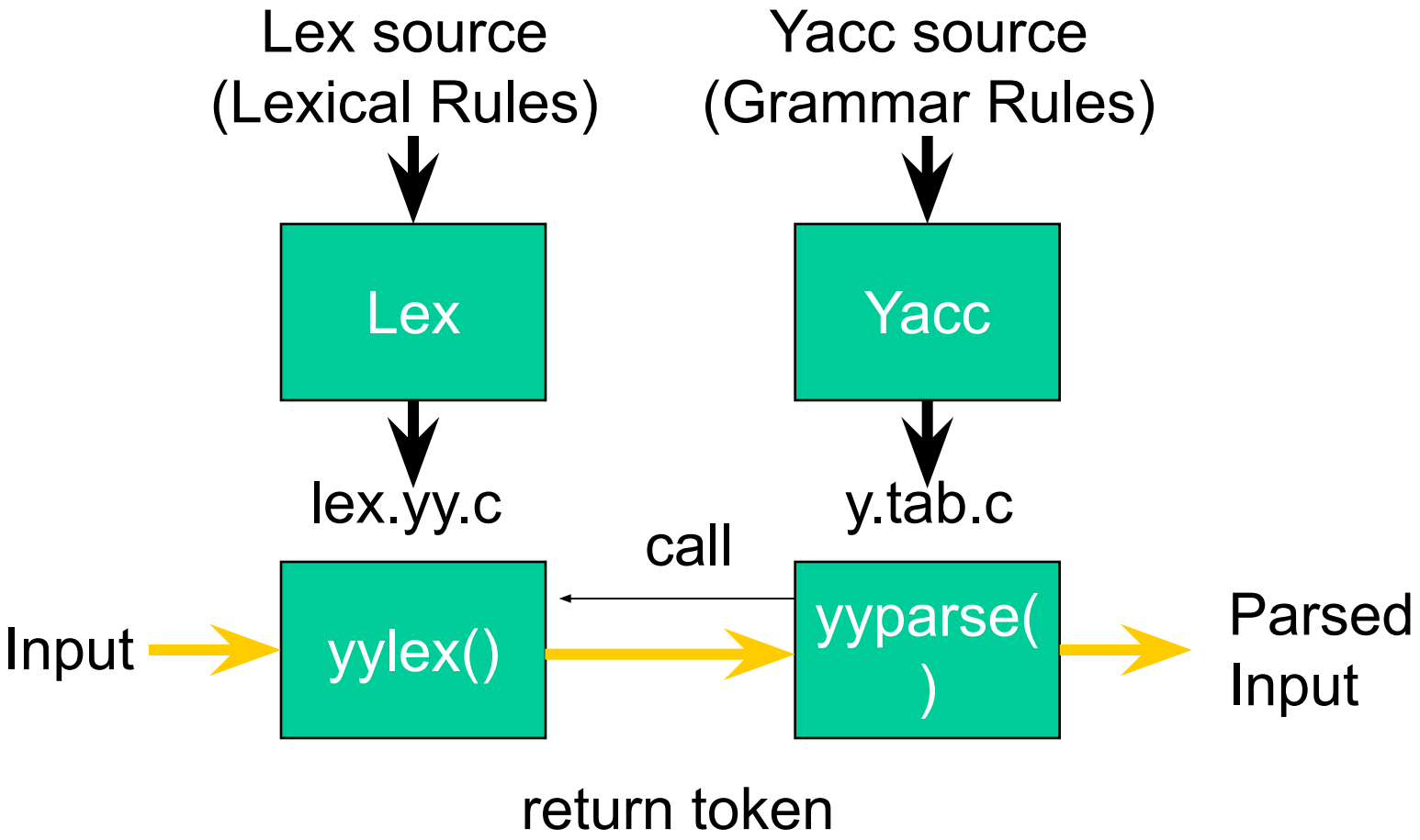
LEX

- Lex is a scanner generator
- Input is description of patterns and actions
- Output is a C program which contains a function yylex() which, when called, matches patterns and performs actions per input
- Typically, the generated scanner performs lexical analysis and produces tokens for the (YACC-generated) parser

Basic Operational Sequence

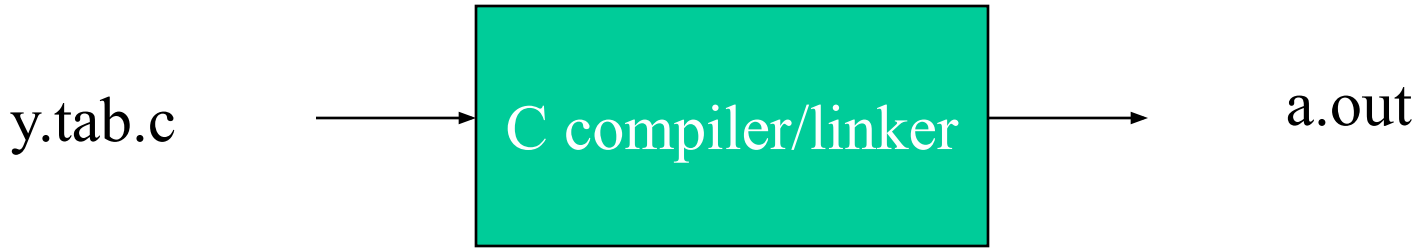
- flex and bison are GNU tools



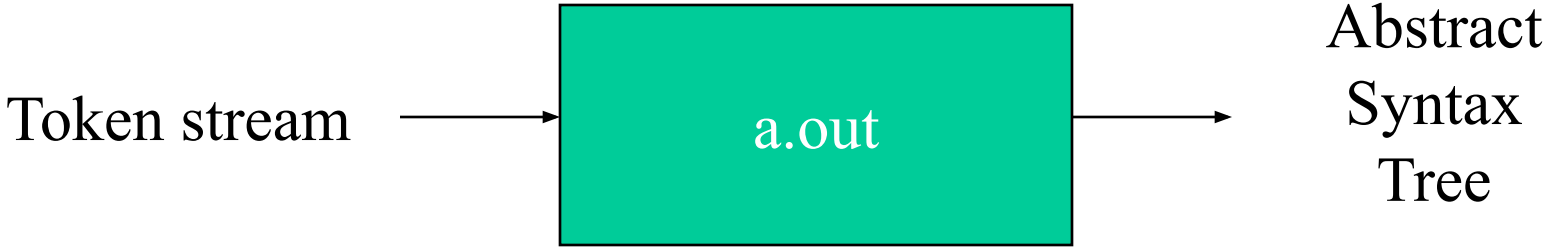

Lex source
(Lexical Rules)
Yacc source
(Grammar Rules)
Lex
Yacc
lex.yy.c
y.tab.c
call
Input
yylex()
yyparse(
)
Parsed
Input
return token

How YACC Works

(1) Parser generation time



(2) Compile time



(3) Run time

YACC Specification section

declarations

%%

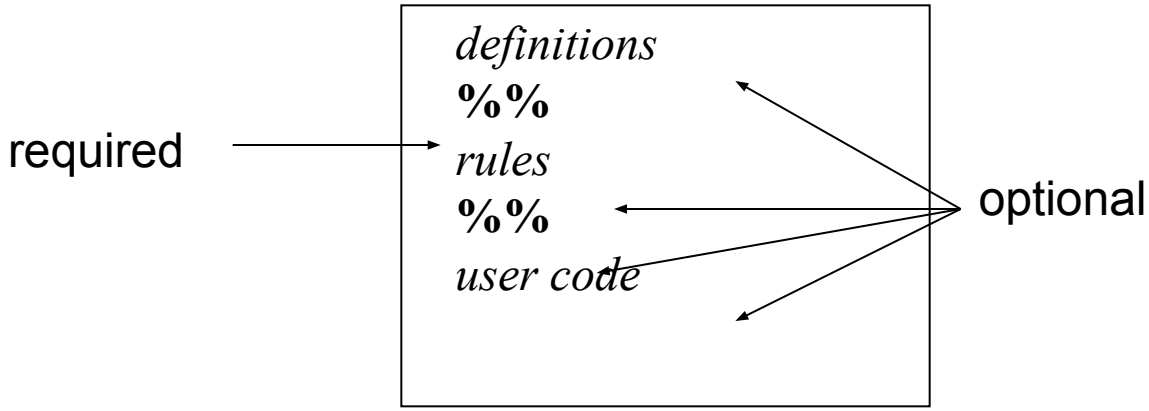
translation rules

%%

Additional C/C++ code

– **Comments enclosed in /* ... */ may appear in any of the sections.**

definitions
%%
rules
%%
user code



Shortest possible legal yacc input:

```
%%
```

YACC declaration section

- Includes:
  - Optional C/C++/Java code (%{ … %}) – copied directly into the output(e.g., declarations, #includes)
  - YACC definitions (%token, %start, …) – used to provide additional information
    - %token – interface to lex
    - %start – start symbol of grammar
    - Others: %type, %left, %right, %union …

YACC rules

- A rule captures all of the productions for a single non-terminal.
  - Left_side : production 1|production 2…..| production n;
- Each rule consists of a grammar production and the associated semantic action.
- The above would be written in Yacc as:

- Left side : production 1 {semantic action 1}
  | production 2 {semantic action 2}
  ……
  | production n {semantic action n}

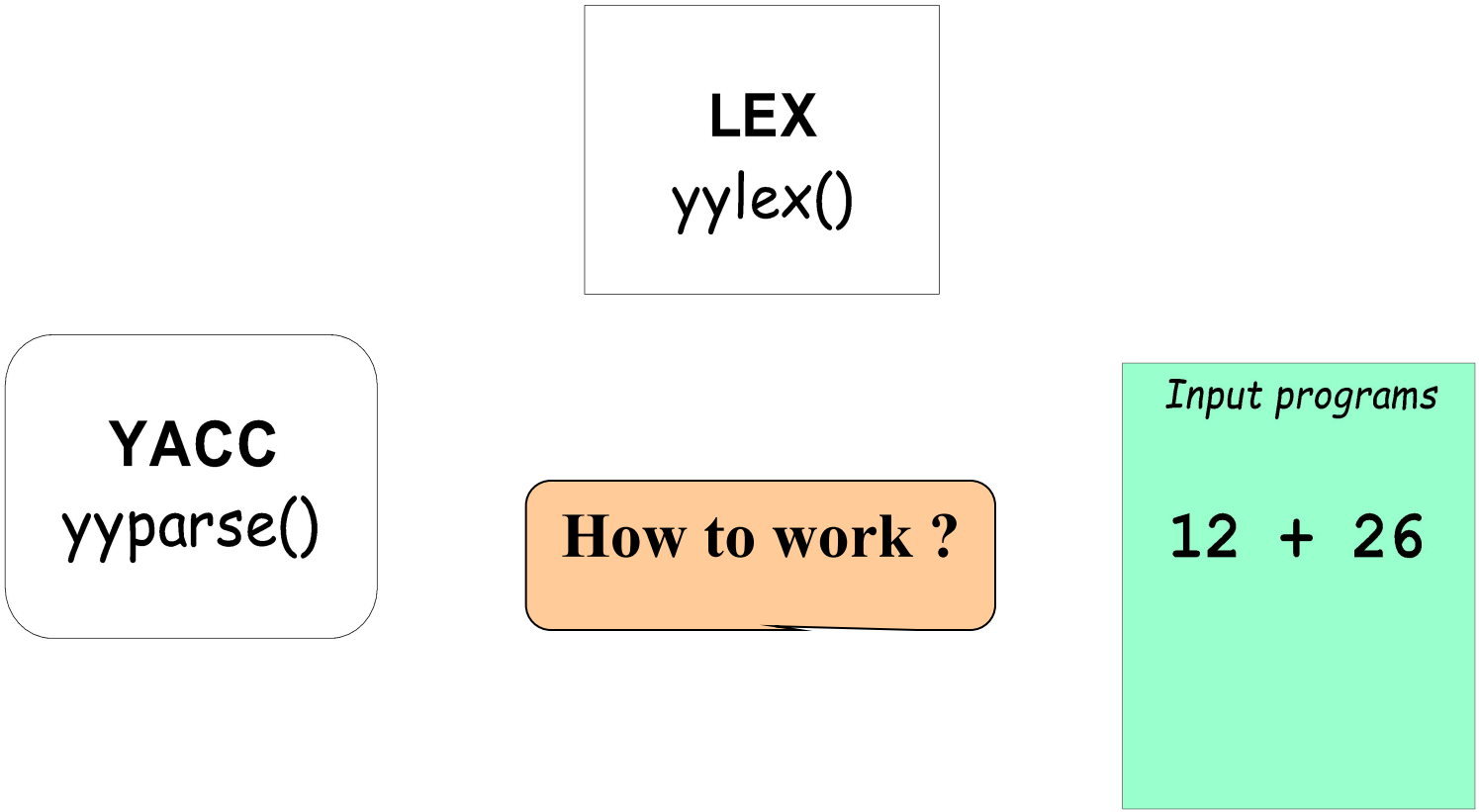
Where Left side is treated as $$, and

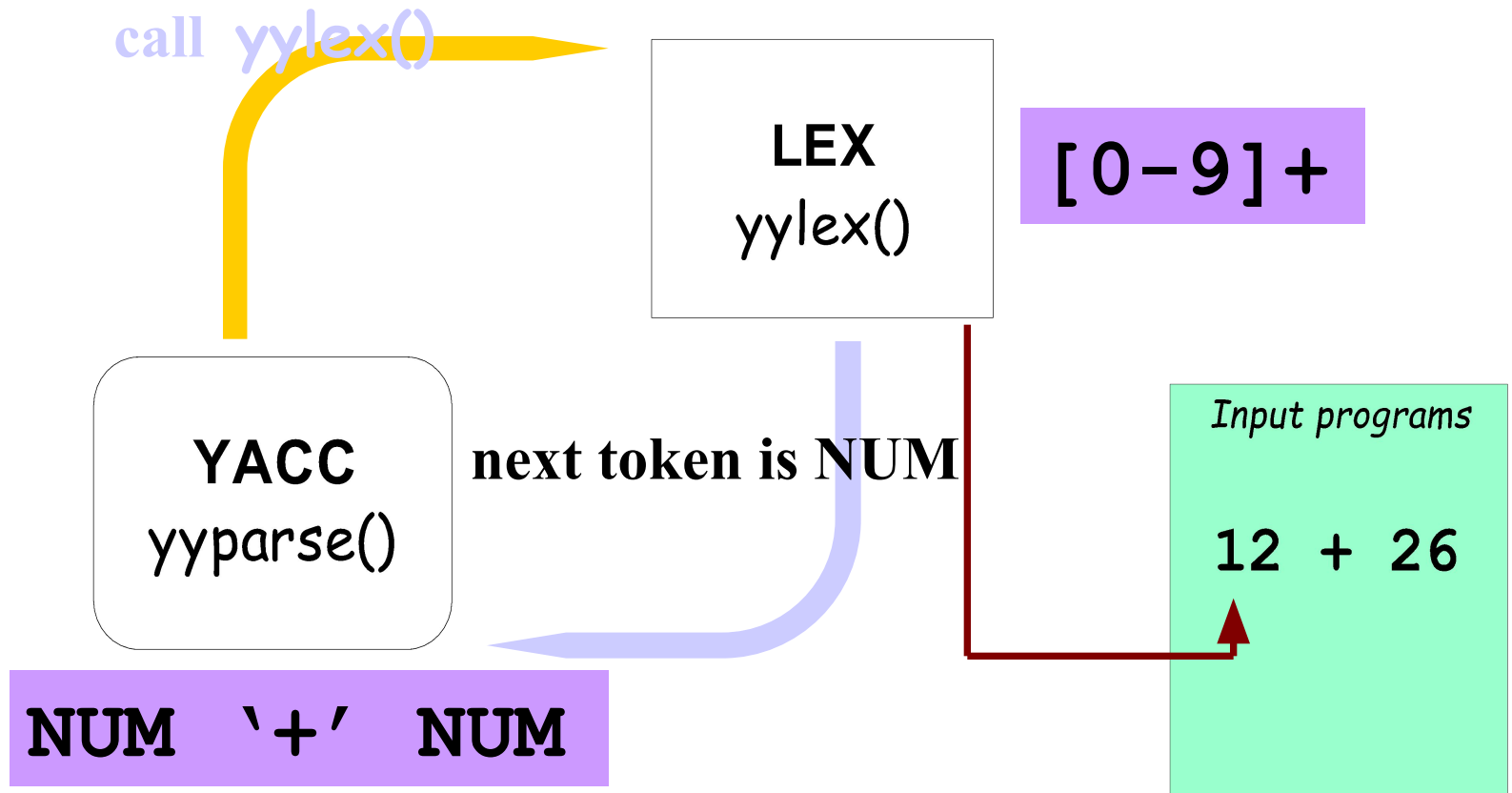
suppose production 1 is x+y, then x can be accessed as $1, + as $2 and y as $3

YACC actions

- Actions are C/C++/Java code.
- Actions can include references to attributes associated with terminals and non-terminals in the productions.
- Actions may be put inside a rule – action performed when symbol is pushed on stack
- Safest (i.e. most predictable) place to put action is at end of rule.

LEX and YACC




call yylex()
LEX
yylex()
[0-9]+
YACC
yyparse()
next token is NUM
Input programs
12 + 26
NUM '+' NUM

Integration with Lex

- *yyparse()* calls *yylex()* when it needs a new token. YACC handles the interface details

In the Lexer:	In the Parser:
return(TOKEN)	%token TOKEN TOKEN used in productions
return('c')	'c' used in productions

- *yylval* is used to return attribute information

- yyparse() reads a stream of token/value pairs from yylex(), which needs to be supplied
- The yylex() as written by Lex reads characters from a FILE * file pointer called yyin. If we do not set yyin, it defaults to standard input. It outputs to yyout, which if unset defaults to stdout.
- We can also modify yyin in the yywrap() function which is called at the end of a file. It allows to open another file, and continue parsing. If this is the case, it will return 0. If we want to end parsing at this file, it returns 1.

- Each call to yylex() returns an integer value which represents a token type. This tells YACC what kind of token it has read. The token may optionally have a value, which should be placed in the variable yylval.
- By default yylval is of type int, but we can override that from the YACC file by re #defining YYSTYPE.
- As yylex() needs to return what kind of token it encountered, and put its value in yylval from yytext. When these tokens are defined with the %token command, they are assigned numerical id's, starting from 257. Because of that fact, it is possible to have all ascii characters as a token.

A simple thermostat controller

heat on

    Heater on!

heat off

    Heater off!

target temperature 22

    New temperature set!

The tokens we need to recognize are: **heat, on/off (STATE), target, temperature, NUMBER.**

thermostat.l

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
%%
[0−9]+          return NUMBER;
heat            return TOKHEAT;
on|off          return STATE;
target          return TOKTARGET;
temperature     return TOKTEMPERATURE;
\n          /* ignore end of line */;
[ \t]+          /* ignore whitespace */;
%%
```

- There are two important changes.
- First, we include the file 'y.tab.h', and secondly, we no longer print stuff, we return names of tokens. This change is because **we are now feeding it all to YACC**, which isn't interested in what we output to the screen. **y.tab.h has definitions for these tokens.**
- But where does y.tab.h come from? It is generated by YACC from the grammar file we are about to create.

grammar rules section of thermostat.y

```
commands: /* empty */
        | commands command
        ;
command:
        heat_switch
        |
        target_set
        ;
heat_switch:
        TOKHEAT STATE
        {
        printf("\tHeat turned on or off\n");
        }
        ;
target_set:
        TOKTARGET TOKTEMPERATURE NUMBER
        {
        printf("\tTemperature set\n");
        }
        ;
```

Header section of thermostat.y

- The previous section only showed the grammar part of the YACC file, but there is more. This is the header that we omitted:

```
%{
#include <stdio.h>
#include <string.h>
void yyerror(const char *str)
{
fprintf(stderr,"error: %s\n",str);
}
int yywrap()
{
return 1;
}
main()
{
yyparse();
}
%}
%token NUMBER TOKHEAT STATE TOKTARGET TOKTEMPERATURE
```

- The yyerror() function is called by YACC if it finds an error.
- The function yywrap() can be used to continue reading from another file. It is called at EOF and you can then open another file, and return 0. Or you can return 1, indicating that this is truly the end.
- Then there is main() function, that does nothing but set everything in motion.
- The last line simply defines the tokens we will be using. These are output using y.tab.h if YACC is invoked with the '−d' option. By which yacc writes an extra output file containing macro definitions for the token type names that are defined in the grammar, the semantic value type YYSTYPE, and a few external variable declarations

Compiling & running the thermostat controller

- lex thermostat.l
- yacc −d thermostat.y
- cc lex.yy.c y.tab.c −o thermostat

- The Lexer needs to be able to access yylval. In order to do so, it must be declared in the scope of the lexer as an extern variable. The original YACC neglects to do this for you, so we should add the following to the lexter, just beneath #include <y.tab.h>:

  extern YYSTYPE yylval;

How to handle parameters

- Whenever Lex matches a target, it puts the text of the match in the character string 'yytext'. YACC in turn expects to find a value in the variable 'yylval'.

```
%{
#include <stdio.h>
#include "y.tab.h“
extern YYSTYPE yylval;
%}
%%
[0−9]+          yylval=atoi(yytext); return NUMBER;
heat            return TOKHEAT;
on|off          yylval=!strcmp(yytext,"on"); return STATE;
target          return TOKTARGET;
temperature     return TOKTEMPERATURE;
\n          /* ignore end of line */;
[ \t]+          /* ignore whitespace */;
%%
```

grammar rules section of thermostat.y

```
commands: /* empty */
        | commands command
        ;
command:
        heat_switch
        |
        target_set
        ;
target_set:
        TOKTARGET TOKTEMPERATURE NUMBER
        {
        printf("\tTemperature set to %d\n",$3);
        }
        ;
heat_switch:
        TOKHEAT STATE                                   {$$ = $1 $2}
        {
                if($2)
                        printf("\tHeat turned on\n");
                else
                        printf("\tHeat turned off\n");
        }
        ;
```

Advanced yylval: %union

- Currently, we need to define the type of yylval. This however is not always appropriate.
- What if we need to handle multiple data types??

- To store only string value to yylval,

```
typedef char* string;
#define YYSTYPE string
```

Modified lexer

```
%{
#include <stdio.h>
#include <string.h>
#include "y.tab.h"
extern YYSTYPE yylval;
%}
 %%
[0-9]+          yylval.number=atoi(yytext); return NUMBER;
heater          return TOKHEATER;
heat            return TOKHEAT;
on|off          yylval.number=!strcmp(yytext,"on"); return STATE;
target          return TOKTARGET;
temperature     return TOKTEMPERATURE;
[a-z0-9]+       yylval.string=strdup(yytext);return WORD;
\n          /* ignore end of line */;
[ \t]+          /* ignore whitespace */;
%%
```

Modified parser

```
%token TOKHEATER TOKHEAT TOKTARGET TOKTEMPERATURE

%union
{
int number;
char *string;
}
%token <number> STATE
%token <number> NUMBER
%token <string> WORD
%%
/* grammar rules */
```

```
heat_switch:
        TOKHEAT WORD
        {
                printf("\tSelected heater '%s'\n",$2);
                heater=$2;  // just to store copy
        };
        |
        TOKHEAT STATE
        {
                printf("\tSelected heater '%d'\n",$2);
                heater=$2;
        };


target_set:
        TOKTARGET TOKTEMPERATURE NUMBER
        {
                printf("\tHeater '%s' temperature set to %d\n",heater,$3);
        }
         ;
```

Features

- Yacc takes a default action when there is a conflict.
- For shift-reduce conflicts yacc will shift. For reduce-reduce conflicts it will use the first rule in the listing.
- It also issues a warning message whenever a conflict exists. The warnings may be suppressed by making the grammar unambiguous.
- You can specify precedence and associativity in YACC, making your grammar simpler.
- Associativity: %left, %right, %nonassoc
- Precedence : order is important (higher at last)

```
%left PLUS MINUS
    %left MULT DIV
    %nonassoc UMINUS
```

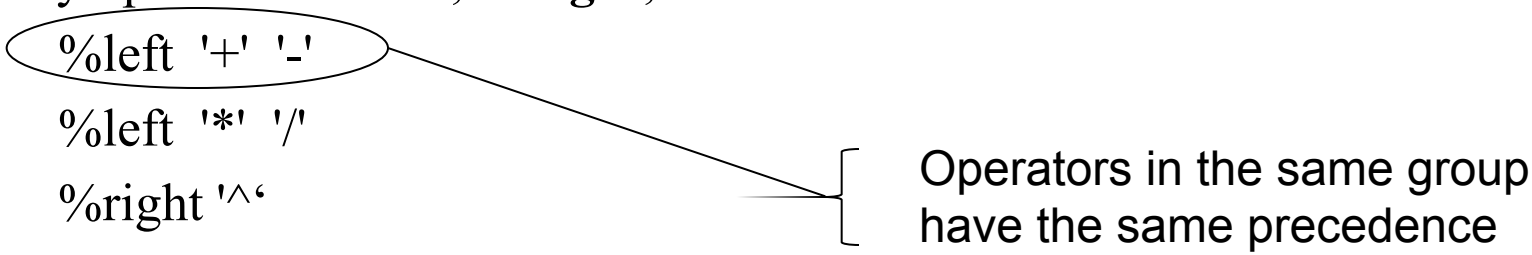
- Lab Assignment – build calculator which can handle basic operations like +,-,*,/,etc using lex and yacc.

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*]
- Removing conflicts:
  - specify operator precedence, associativity;
  - restructure the grammar
    - use **y.output** to identify reasons for the conflict.

Specifying Operator Properties

- Binary operators: **%left**, **%right**, **%nonassoc**:



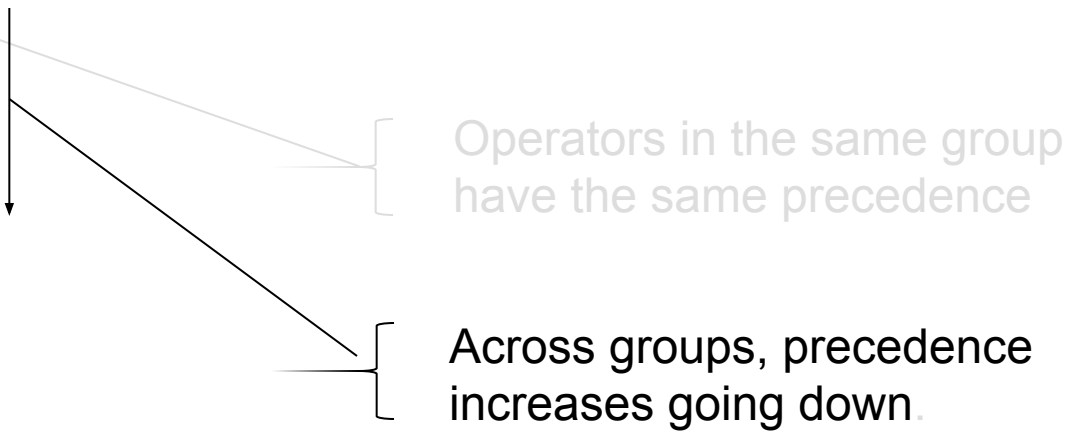
- Unary operators: **%prec**
  - Changes the precedence of a rule to be that of the token specified. E.g.:

```
%left '+' '-'
%left '*' '/'
Expr: expr '+' expr
    | '–' expr   %prec '*'
    | …
```

Specifying Operator Properties

- Binary operators: **%left**, **%right**, **%nonassoc**:

```
%left  '+'  '-'
%left  '*'  '/'
%right '^'
```



- Unary operators: **%prec**
  - Changes the precedence of a rule to be that of the token specified. E.g.:

```
%left '+'  '-'
%left  '*'  '/'
Expr: expr ‘+’ expr
      | ‘–’ expr    %prec ‘*’
      | …
```

Specifying Operator Properties

- Binary operators: **%left**, **%right**, **%nonassoc**:

```
%left  '+'  '-'
%left  '*'  '/'
%right '^'
```

Operators in the same group have the same precedence

Across groups, precedence increases going down.

- Unary operators: **%prec**
  - Changes the precedence of a rule to be that of the token specified. E.g.:

```
%left '+'  '-'
%left  '*'  '/'
Expr: expr '+' expr
      | '–' expr    %prec '*'
      | …
```

The rule for unary '–' has the same (high) precedence as '*'

Example

- The grammar:

  program -> program expr | ε

  expr -> expr + expr | expr - expr | id

- Program and expr are nonterminals.
- id are terminals (tokens returned by lex) .

- expression may be :
  - sum of two expressions
  - product of two expressions
  - Or an identifiers

Lex file

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "y.tab.h"
%}

%%

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

[-+\n]       return *yytext;

[ \t]        ; /* skip whitespace */

.            yyerror("invalid character");

%%

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

Yacc file

```
%{
    #include <stdio.h>
    int yylex(void);
    void yyerror(char *);
%}

%token INTEGER

%%

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

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

%%

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

int main(void) {
    yyparse();
    return 0;
}
```

Linking lex & yacc