Cairo University
Faculty of Engineering
Computer Engineering Department
CMPN403 Languages and Compilers

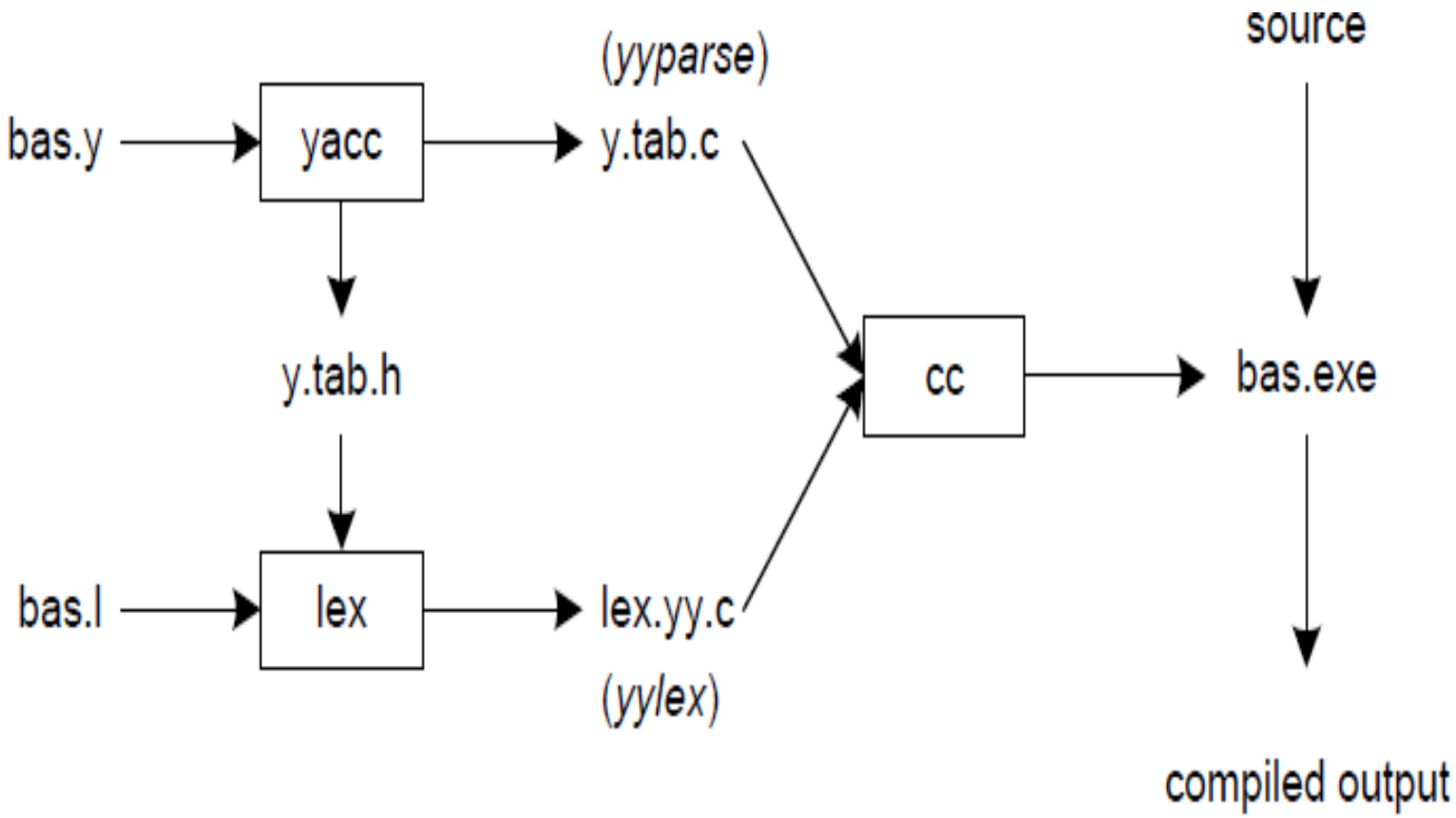
Lex and Yacc

LYDIA WAHID AMIN

Installation

Flex and bison, clones for lex and yacc, can be obtained for free from GNU and Cygwin.

Introduction



Introduction

First, we need to specify all pattern matching rules for lex (**bas.l**) and grammar rules for yacc (**bas.y**). Commands to create our compiler, **bas.exe**, are listed below:

```
yacc -d bas.y                  # create y.tab.h, y.tab.c
lex bas.l                      # create lex.yy.c
cc lex.yy.c y.tab.c -obas.exe  # compile/link
```

Introduction

- Yacc reads the grammar descriptions in **bas.y** and generates a syntax analyzer (parser), that includes function **yyparse**, in file **y.tab.c**.
- Included in file **bas.y** are token declarations. The **–d** option causes yacc to generate definitions for tokens and place them in file **y.tab.h**.
- Lex reads the pattern descriptions in **bas.l**, includes file **y.tab.h**, and generates a lexical analyzer, that includes function **yylex**, in file **lex.yy.c**.
- Finally, the lexer and parser are compiled and linked together to create executable **bas.exe**.
- From **main** we call **yyparse** to run the compiler. Function **yyparse** automatically calls **yylex** to obtain each token.

Lex

Input to Lex is divided into three sections with **%%** dividing the sections.

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

Lex - Example

```
%{
    int nchar, nword, nline;
%}
%%
\n          { nline++; nchar++; }
[^ \t\n]+   { nword++, nchar += yyleng; }
.           { nchar++; }
%%
int main(void) {
    yylex();
    printf("%d\t%d\t%d\n", nchar, nword, nline);
    return 0;
}
```

length of matched string

call to invoke lexer, returns token

bas.l

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

#include "y.tab.h"
%}

%%

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

[-+\n]          return *yytext;

[ \t]           ; /* skip whitespace */



%%

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

value associated with token

pointer to matched string

called by lex when input is exhausted. Return 1 if you are done

Yacc

Input to yacc is divided into three sections.

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

The definitions section consists of token declarations and C code bracketed by "**%{**" and "**%}**".

The grammar is placed in the rules section and user subroutines are added in the subroutines section.

bas.y

```
%{
    #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;
}
```

Yacc

- Internally yacc maintains two stacks in memory; a parse stack and a value stack.
- The parse stack contains terminals and nonterminals that represent the current parsing state.
- The value stack is an array of elements and associates a value with each element in the parse stack.
- For example when lex returns an **INTEGER** token yacc shifts this token to the parse stack. At the same time the corresponding **yylval** is shifted to the value stack.
- The parse and value stacks are always synchronized so finding a value related to a token on the stack is easily accomplished.

Yacc

- We replace the right-hand side of the production in the parse stack with the left-hand side of the same production. For example, we pop "**expr '+' expr**" and push "**expr**".
- We have reduced the stack by popping three terms off the stack and pushing back one term.
- "**$$**" designates the top of the stack after reduction has taken place. The above action adds the value associated with two expressions, pops three terms off the value stack, and pushes back a single sum.
- As a consequence the parse and value stacks remain synchronized.