



UNIVERSIDAD DE  
GUANAJUATO

Universidad de Guanajuato  
División de Ingenierías Campus Irapuato Salamanca  
Compiladores  
Diego Eduardo Rosas González

## Práctica #6: Diseño de un Compilador

Objetivo: Diseñar un compilador usando Flex y Bison.

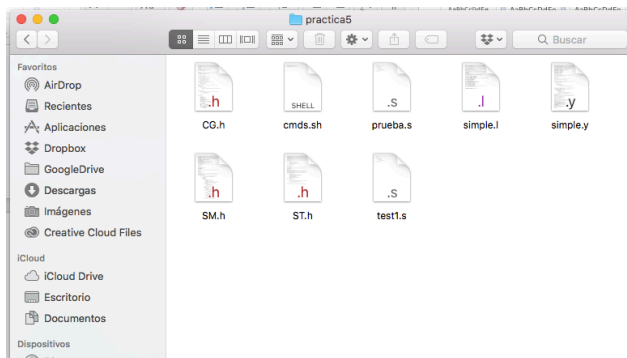
### Desarrollo

1. En caso de que no tener instalado Cygwin, instálelo con los paquetes gcc, bison y flex.

```
diego con los paquetes gcc,
1. diego@MacBook-Pro-de-Diego-2: ~ (zsh)
X diego@paperLessDr... %1 X ~ (zsh) %2 X ~ (zsh) %3
Last login: Tue Dec 4 00:08:26 on ttys000
→ ~ bison -v
bison: missing operand
Try 'bison --help' for more information.
→ ~ bison --version
bison (GNU Bison) 3.2
Written by Robert Corbett and Richard Stallman.

Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
→ ~ flex --version
flex 2.5.35 Apple(flex-31)
```

2. Descargue de la carpeta compartida del curso la carpeta practica5.

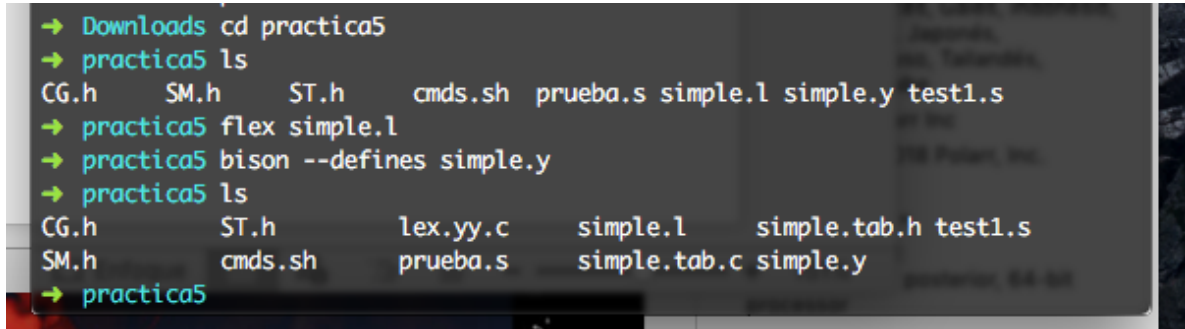


**3. Compile el archivo simple.l.**

```
flex simple.l
```

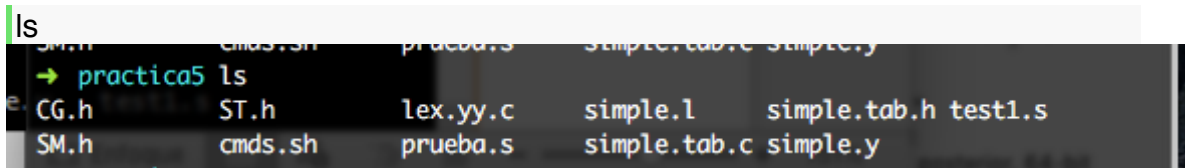
**4. Compile el archivo simple.y.**

```
bison --defines simple.y
```



```
→ Downloads cd practica5
→ practica5 ls
CG.h    SM.h    ST.h    cmds.sh  prueba.s  simple.l  simple.y  test1.s
→ practica5 flex simple.l
→ practica5 bison --defines simple.y
→ practica5 ls
CG.h    ST.h    lex.yy.c  simple.l  simple.tab.h  test1.s
SM.h    cmds.sh  prueba.s  simple.tab.c  simple.y
```

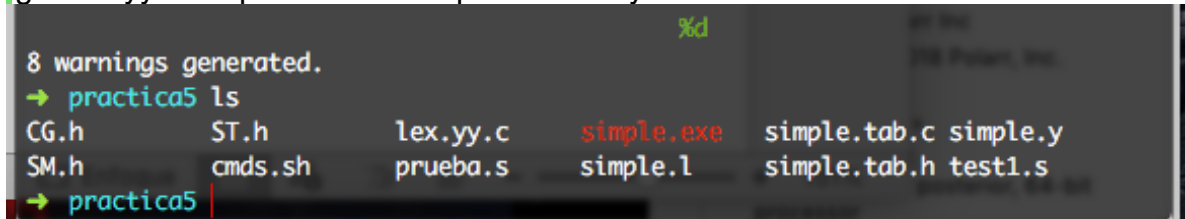
**5. Liste los archivos generados**



```
ls
→ practica5 ls
CG.h    ST.h    lex.yy.c  simple.l  simple.tab.h  test1.s
SM.h    cmds.sh  prueba.s  simple.tab.c  simple.y
```

**6. Compile el código fuente generado para obtener el ejecutable del compilador.**

```
gcc lex.yy.c simple.tab.c -o simple.exe -lfl -ly -lm
```



```
8 warnings generated.
→ practica5 ls
CG.h    ST.h    lex.yy.c  simple.exe  simple.tab.c  simple.y
SM.h    cmds.sh  prueba.s  simple.l    simple.tab.h  test1.s
→ practica5
```

**7. Pruebe el ejecutable con archivos de texto en formato Unix que contenga programas completos.**

```
./simple.exe prueba.s
```

```
1. diego@MacBook-Pro-de-Diego-2: ~/Downloads/practica5 (zsh)
→ practica5 ./simple.exe prueba.s
Codigo Tokenizado:
PROGRAM VAR INT ID(1) , ID(suma) ; BEGIN ID(1) := NUMBER(0) ; ID(suma) := NUMBER
(0) ; WHILE ID(1) < NUMBER(10) DO ID(suma) := ID(suma) + ID(1) ; ID(1) := ID(1)
+ NUMBER(1) ; END ; PRINT ID(suma) ; END

Compilacion finalizada exitosamente!
Codigo Intermedio:
Line opcode operand
0: data 2
1: ld_int 0
2: store 0
3: ld_int 0
4: store 1
5: ld_var 0
6: ld_int 10
7: lt 0
8: jmp_false L8
9: ld_var 1
10: ld_var 0
11: add 0
12: store 1
13: ld_var 0
14: ld_int 1
15: add 0
16: store 0
17: goto 5
18: ld_var 1
19: out_int 0
20: halt 0

Codigo ensamblador:
.s56
.model flat,c
print PROTO C :VARARG ;
.DATA 0x100
.DATA
```

```
1. diego@MacBook-Pro-de-Diego-2: ~/Downloads/practica5 (zsh)
→ practica5 ./simple.exe prueba.s
Codigo Tokenizado:
PROGRAM VAR INT ID(1) , ID(suma) ; BEGIN ID(1) := NUMBER(0) ; ID(suma) := NUMBER
(0) ; WHILE ID(1) < NUMBER(10) DO ID(suma) := ID(suma) + ID(1) ; ID(1) := ID(1)
+ NUMBER(1) ; END ; PRINT ID(suma) ; END

Compilacion finalizada exitosamente!
Codigo Intermedio:
Line opcode operand
0: data 2
1: ld_int 0
2: store 0
3: ld_int 0
4: store 1
5: ld_var 0
6: ld_int 10
7: lt 0
8: jmp_false L8
9: ld_var 1
10: ld_var 0
11: add 0
12: store 1
13: ld_var 0
14: ld_int 1
15: add 0
16: store 0
17: goto 5
18: ld_var 1
19: out_int 0
20: halt 0

Codigo ensamblador:
.s56
.model flat,c
print PROTO C :VARARG ;
.DATA 0x100
.DATA

Ejecutando codigo intermedio:
Salida: 45
halt
→ practica5
```

8. Modifique la gramática del compilador anterior (regla declarations) para que pueda haber más de una línea de declaración de variables.

```

;
header: /* empty */
      | VAR declarations
;
declarations : /* empty */
      | INT id_seq ID ';' { install( $3 ); }
;
id_seq : /* empty */
      | id_seq ID ',' { install( $2 ); }
;
commands : /* empty */
      | commands command ';'
;
command : NOP
      | INPUT ID { context_check( READ_INT, $2 ); }
      | PRINT expr { gen_code( WRITE_INT, 0 ); }
      | ID ASSGNOP expr { context_check( STORE, $1 ); }
      | IF expr { $1 = (struct lbs *) newlblrec();
                  $1->for_jump_false = reserve_loc();
                  THEN commands { $1->for_goto = reserve_loc(); }
                  ELSE { back_patch( $1->for_jump_false,

```

9. Modifique la gramática del compilador anterior para que se puedan hacer operaciones lógicas AND, OR y NOT. Defina los tokens y sus precedencias. Agregue las reglas para detectar los tokens al archivo de léxico.

```

%}
/*=====
TOKEN Definitions
=====*/
DIGIT [0-9]
ID [a-z][a-z0-9]*
/*=====
REGULAR EXPRESSIONS defining the tokens for the Simple language
=====*/
%%
"=" { printf("%s ",yytext);return(ASSGNOP); }
"<=" { printf("%s ",yytext);return(LTEOP); }
">=" { printf("%s ",yytext);return(GTEOP); }
"!=" { printf("%s ",yytext);return(NEOP); }
{DIGIT}+ { printf("NUMBER(%s) ",yytext);yylval.intval = atoi( yytext );return(NUMBER); }
do { printf("DO ",yytext);return(DO); }
else { printf("ELSE ");return(ELSE); }
end { printf("END ");return(END); }
if { printf("IF ");return(IF); }
begin { printf("BEGIN ");return(BEGINP); }
int { printf("INT ");return(INT); }
program { printf("PROGRAM ",yytext);return(PROGRAM); }
input { printf("INPUT ",yytext);return(INPUT); }
nop { printf("NOP ");return(NOP); }
then { printf("THEN ");return(THEN); }
while { printf("WHILE ");return(WHILE); }
print { printf("PRINT ");return(PRINT); }
var { printf("VAR ");return(VAR); }
and {printf("AND ");return(AND);}
or {printf("OR ");return(OR);}
not {printf("NOT ");return(NOT);}
[ID] { printf("ID(%s) ",yytext); yylval.id = (char *) strdup(yytext);return(ID); }
[ \t\n]+ /* eat up whitespace */{ }
. { printf("%c ",*yytext);return(yytext[0]); }
%%

```

```

int intval; /* Integer values */
char *id; /* Identifiers */
struct lbs *lbs; /* For backpatching */
}
/*=====
TOKENS
=====*/
%start program
%token <intval> NUMBER /* Simple integer */
%token <id> ID /*
%token <lbls> IF \
%token NOP THEN EI
%token INT INPUT ;
%token ASSGNOP
/*=====
OPERATOR PRECEDENCE
=====
%left '>' '=' '<'
%left '-' '+'
%left '*' '/'
%right NEG
/*=====
GRAMMAR RULES for
=====
%%
program : PROGRAM

;
header: /* empty
        | VAR decl
        ;
declarations : /*

=====
/* OPERATIONS: In
enum code_ops { H
DATA, LD_INT, LD_
READ_INT, WRITE_I
LT, LTE, EQ, NEQ,
/* OPERATIONS: Ex
char *op_name[] =
"data", "ld_int",
"in_int", "out_in
"lt", "lte", "eq"
struct instruction
{
enum code_ops op;
int arg;
};
/* CODE Array */
struct instruction code[999];
/* RUN-TIME Stack */
int stack[999];

```

10. Repita los pasos anteriores para probar sus modificaciones.

```

Codigo Tokenizado:
PROGRAM VAR INT ID(i) , ID(suma) ; BEGIN ID(i) := NUMBER(0) ; ID(suma) := NUMBER
(0) ; WHILE ID(i) < NUMBER(10) DO ID(suma) := ID(suma) + ID(i) ; ID(i) := ID(i)
+ NUMBER(1) ; END ; PRINT ID(suma) ; END

compilacion finalizada exitosamente!

Codigo intermedio:
_line opcode operand
0: data 2
1: ld_int 0
2: store 0
3: ld_int 0
4: store 1
5: ld_var 0
6: ld_int 10
7: lt 0
8: jmp_false 18
9: ld_var 1
10: ld_var 0
11: add 0
12: store 1
13: ld_var 0

```