

Construcción de un interprete en Lex/Yacc.

Listings

1	Implementación del scanner en LEX	5
2	Implementación de los nodos del árbol	9
3	Implementación del Parser en YACC	10

List of Figures

1	Fases de un compilador	2
2	Accesos a la tabla de símbolos por parte de las distintas fases de un compilador	8
3	Construcción del interprete	15
4	Prueba de operaciones aritméticas	16
5	Condiciones iterativas	16

List of Tables

1	Tokens utilizados en el lenguaje	4
2	Gramática en BNF	7

Esta versión de un lenguaje de programación creado no resultara ser más compleja que otras versiones anteriores. Además de esto, se construirá un árbol de sintaxis durante el análisis.

Después de analizar, recorreremos el árbol de sintaxis para producir resultados. Se podrán aplicar dos versiones del recorrido del árbol, la cuales se presentarán a continuación:

1. Un interprete que ejecuta las declaraciones durante el recorrido del árbol.
2. Un compilador que genera código para una maquinaria hipotética basada en una pila.

1 Introducción

El trabajo de un compilador es traducir lenguaje puro de alto nivel en código ensamblador. Este compilador se comprende por 6 fases, en la Figura 1 se muestran estas fases.

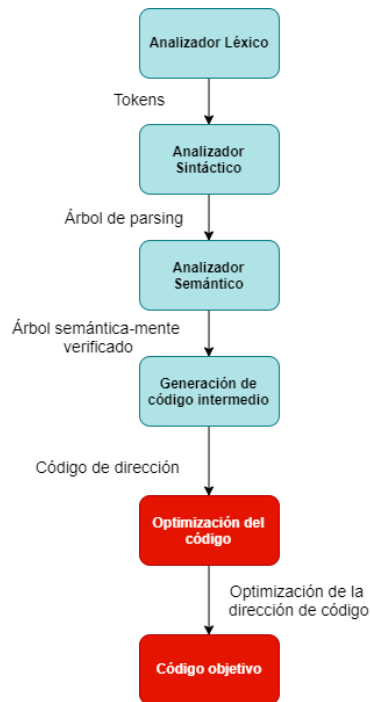


Figure 1: Fases de un compilador

1.1 Interfaz

En esta parte, hace que el compilador sea independiente de la arquitectura del sistema. Se compone de las siguientes fases:

1. Analizador léxico: Aquí se identifican los **lexemas** en el archivo, en este archivo se coincide con las expresiones regulares en el escáner para generar **tokens** para el analizador. Aquí también se eliminan los espacios en blanco y los comentarios.
2. Analizador sintáctico: Toma los tokens generados por el analizador léxico, después verifica la sintaxis usando **CFG** (*Gramáticas libres de contexto*) y genera un **árbol de análisis**.
3. Analizador semántico: Aquí se comprueba si el código es significativo y coherente. Al igual que la resolución de alcance, la verificación de tipo y la verificación de matriz.

4. Generación de código intermedio: Obtenido ya el árbol de análisis semánticamente verificado, se procede a generar el código de tres direcciones (o cualquier otro código independiente de la plataforma).

1.2 Backend

En esta sección, se genera un código de ensamblaje optimizado a partir del código intermedio y, por lo tanto, es independiente del lenguaje de programación de alto nivel. Las partes que lo comprende, son las siguientes:

1. Optimización de código: Aquí se emplean diferentes técnicas de optimización para tratar de optimizar la sección del código intermedia.
2. Generación de código: Genera código de acuerdo con la arquitectura del sistema a partir del código intermedio optimizado.

2 Analizador léxico

2.1 Introducción al analizador léxico

El escáner se implemento con la herramienta de *Lex*. Este lee la entrada de un archivo y la compara con las expresiones regulares para generar los tokens correspondientes como se muestra en la Tabla 1.

2.2 Tokens

Tokens	Exresión Regular
IF, ELSE	if, else
WHILE	while
PRINT	print
VAR	[a-z]
INTEGER	[0 - 9][0 - 9]*
GE, LE, EQ, NE, <,>	>=, <=, ==, !=, <,>
+, -, *, /, %	+, -, *, /, %
{}, ()	{}, ()
=	=
DELIMITADOR	;
UMINUS	-

Table 1: Tokens utilizados en el lenguaje

2.3 Implementación

Para la implementación, los tokens se han definido en el archivo *y.tab.h*. La instrucción *yytext* denota el lexema que coincide con el patrón.

El atributo asociado con **VAR** es *yyval.idx*, que denota el índice de esa variable en la tabla de símbolos. El atributo asociado con **INTEGER** es *yyval.val*, el cual denota el valor de ese número.

El escáner se implementó utilizando el siguiente código en *Lex*.

```
1 %{
2     #include <stdio.h>
3     #include y.tab.h
4     void yyerror (char*);
5 %}
6
7
8 letter [a-zA-Z_]
9 digit [0-9]
10
11 %%
12
13 if      {return IF;}
14 else    {return ELSE;}
15 while   {return WHILE;}
16 print   {return PRINT;}
17
18 {letter}({letter}|{digit})*    {yyval.idx = *yytext - 'a'; return VAR;}
19
20 0      {yyval.val = atoi(yytext); return INTEGER;}
21 [1-9][0-9]*    {yyval.val = atoi(yytext); return INTEGER;}
22
23 [-+<>{}()/%*]=    {return *yytext;}
24
25 <=      {return LE;}
26 >=      {return GE;}
27 ==      {return EQ;}
28 !=      {return NE;}
29
30 [ \t]+      ;
31
32 ;      {return DELIM;}
33 \n      {yylineno++;}
34
35 .      yyerror(Unknown literal);
36
37 %%
38
39 int yywrap ()
40 {
41     return 1;
42 }
```

Listing 1: Implementación del scanner en LEX

3 Parser

3.1 Introducción al parser

Todo lenguaje de programación obedece a unas reglas que describen la estructura sintáctica de los programas bien formados que acepta. Se puede describir la sintaxis de las construcciones de los lenguajes de programación por medio de gramáticas de contexto libre o utilizando la notación BNF (*Backus-Naur Form*).

3.2 Gramática

La construcción de los tokens: **IF**, **ELSE** en esta gramática es ambigua y causa un cambio que reduce el conflicto. Cada vez que se produce un cambio de reducción de conflictos, *YACC* elige un cierto de *cambio* en vez de una *reducción*, por lo tanto, **IF**, **ELSE** tiene una mayor prioridad sobre la construcción **IF**.

Tanto el operador UMINUS como el binario menos «−» tienen el mismo patrón «−». Pero se distinguen en *YACC* ya que tienen diferente asociatividad y precedencia. La gramática de la Tabla 2, se encuentra escrita en la forma de *Backus-Naur*.

<program> ::= <block>
<block> ::= <block> <statement> NULL
<statement> ::= DELIM <expr> DELIM PRINT <expr> DELIM VAR '=' <expr> DELIM WHILE '(' <expr> ')' <statement> IF '(' <expr> ')' <statement> IF '(' <expr> ')' <statement> ELSE <statement> " <list> "
<list> ::= <list> <statement> <statement>
<expr> : INTEGER VAR <expr> '+' <expr> <expr> '-' <expr> <expr> '*' <expr> <expr> '/' <expr> <expr> '%' <expr> <expr> '<' <expr> <expr> '>' <expr> <expr> LE <expr> <expr> GE <expr> <expr> EQ <expr> <expr> NE <expr> '-' <expr> %prec UMINUS '(' <expr> ')'

Table 2: Gramática en BNF

3.3 Implementación del parser

3.3.1 Estructura del nodo y la tabla de símbolos

Según [Rojas and Mata \(2005\)](#): «También llamada **tabla de nombres**, se trata sencillamente de una estructura de datos de alto rendimiento que almacena toda la información necesaria sobre los identificadores de usuario». Esta tabla tiene dos funciones principales:

1. Efectuar chequeos semánticos.
2. Generar código.

3.3.2 Consideraciones sobre la tabla de símbolos

Esta tabla puede inicializarse con cierta información útil, esta puede almacenarse en una única estructura o en varias, las cuales se presentan a continuación:

1. Constantes: π , e , etc.
2. Funciones de librería: EXP, LOG, SQRT.
3. Palabras reservadas: Según [Rojas and Mata \(2005\)](#): «Algunos analizadores lexicográficos no reconocen directamente las palabras reservadas, sino que sólo reconocen identificadores de usuario. Una vez tomado uno de la entrada, lo buscan en la tabla de palabras reservadas por si coincide con alguna; si se encuentra, devuelven al analizador sintáctico el token asociado en la tabla; si no, lo devuelven como identificador de verdad. Esto facilita el trabajo al lexicográfico, es más, dado que esta tabla es invariable, puede almacenarse en una tabla de dispersión perfecta».

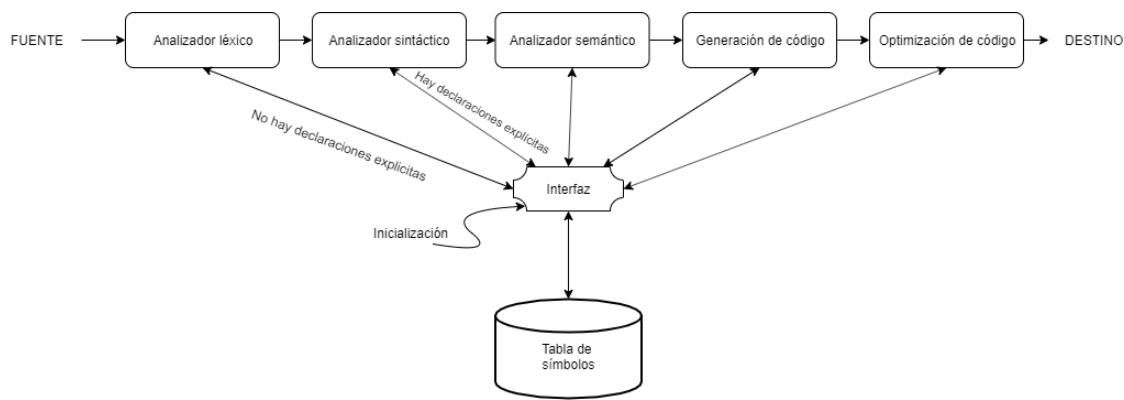


Figure 2: Accesos a la tabla de símbolos por parte de las distintas fases de un compilador

Como se muestra en la Figura 2, y según [Rojas and Mata \(2005\)](#): «conforme avanza la etapa de análisis y van apareciendo nuevas declaraciones de identificadores, el analizador léxico o el analizador sintáctico según la estrategia que sigamos, insertará nuevas entradas en la tabla de símbolos, evitando siempre la existencia de entradas repetidas». El analizador semántico efectúa ciertas comprobaciones sensibles al contexto con ayuda de la tabla de símbolos, y dentro de la etapa de síntesis, el generador de código intermedio implementa las direcciones de memoria asociadas a cada identificador para generar un programa equivalente al de entrada. Según [Rojas and Mata \(2005\)](#): «Por regla general el optimizador de código no necesita hacer uso de ella, aunque nada impide que pueda accederla».

Según [Niemann \(2004\)](#): «La tabla de símbolos, **sym**, permite nombres de variables de un solo carácter. Un nodo en el árbol de sintaxis puede contener una constante (**conNodeType**), un identificador (**idNodeType**) o un nodo interno con un operador (**oprNodeType**). Una

*unión encapsula las tres variantes, y **nodeType.type** se usa para determinar qué estructura tenemos».*

La estructura del nodo es la siguiente:

```
1 typedef enum {type_id, type_const, type_op} node_enum;
2
3 typedef struct
4 {
5     int value;
6 }const_node;
7
8 typedef struct
9 {
10    int index;
11 }id_node;
12
13 typedef struct
14 {
15     int opr;
16     int nop;
17     struct nodetag *op[1];
18 }opr_node;
19
20 typedef struct nodetag
21 {
22     node_enum node_type;
23
24     union
25     {
26         const_node cn;
27         id_node in;
28         opr_node on;
29     };
30 }node;
31
32 extern int symtable[26];
```

Listing 2: Implementación de los nodos del árbol

El archivo de entrada lex contiene patrones para tokens

1. Variables.
2. Enteros.

Según [Rojas and Mata \(2005\)](#): «Las acciones semánticas usarán este puntero para acceder al valor de cada variable: si el identificador está a la izquierda del token de asignación, entonces se machacará el valor; y si forma parte de una expresión, ésta se evaluará al valor de la variable».

Además, los tokens se definen para operadores de 2 caracteres como **EQ** y **NE**. Los operadores de un solo carácter simplemente se devuelven como ellos mismos. El archivo de entrada yacc define *YYSTYPE*, el tipo de *yylval*, como se muestra a continuación:

```
1 %union{
2     int iValue;      /* valor entero */
3     char sIndex;     /* indice de la tabla de simbolos */
4     nodeType *nPtr;  /* apuntador del nodo */
5 };
```

Esto hace que se genere lo siguiente en el archivo de YACC (*y.tab.h*):

```
1 typedef union {
2     int iValue;      /* valor entero */
3     char sIndex;     /* indice de la tabla de simbolos */
4     nodeType *nPtr;  /* apuntador del nodo */
5 } YYSTYPE;
6 extern YYSTYPE yylval;
```

```
1 %{
2     #include node.h
3     #include <stdio.h>
4     #include <stdlib.h>
5     #include <stdarg.h>
6
7     int yylex(void);
8     node* con(int);
9     node* id(int);
10    node* opr(int, int, ...);
11    void free_node (node*);
12    int ex(node*);
13    void yyerror(char*);
14    int symtable[26];
15 %}
16
17
18 %union
```

```

19 {
20     int idx;
21     int val;
22     struct nodetag* nptr;
23 };
24
25 %token <val> INTEGER
26 %token <idx> VAR
27
28 %token IF WHILE PRINT
29 %token DELIM
30
31 %nonassoc IFX
32 %nonassoc ELSE
33
34 %right '='
35 %left '<' '>' LE GE EQ NE
36 %left '+' '-'
37 %left '/' '*' '%'
38 %left '(' ')'
39 %left '{' '}'
40 %nonassoc UMINUS
41
42 %type <nptr> statement list expr
43
44 %%
45
46 program:
47     block    {exit(0);}
48     ;
49
50 block:
51     block statement    {ex($2); free_node($2);}
52     |
53     ;
54
55 statement:
56     DELIM {$$ = opr(';', 2, NULL, NULL);}
57     | expr DELIM      {$$=$1;}
58     | PRINT expr DELIM {$$ = opr(PRINT,1, $2);}
59     | VAR '=' expr DELIM      {$$ = opr('=', 2, id($1), $3);}
60     | WHILE '(' expr ')' statement    {$$ = opr(WHILE, 2, $3, $5);}
61     | IF '(' expr ')' statement %prec IFX    {$$ = opr(IF, 2, $3, $5);}
62     | IF '(' expr ')' statement ELSE statement    {$$ = opr(IF, 3, $3, $5,
63         $7);}
64     | '{' list '}'      {$$ = $2;}
65     ;
66
67 list:
68     list statement    {$$ = opr(';', 2, $1, $2);}

```

```

68 | statement    {$$ = $1;}
69 ;
70
71 expr:
72   INTEGER    {$$ = con($1);}
73 | VAR        {$$ = id($1);}
74 | expr '+' expr {$$ = opr('+', 2, $1, $3);}
75 | expr '-' expr {$$ = opr('-', 2, $1, $3);}
76 | expr '*' expr {$$ = opr('*', 2, $1, $3);}
77 | expr '/' expr {$$ = opr('/', 2, $1, $3);}
78 | expr '%' expr {$$ = opr('%', 2, $1, $3);}
79 | expr '<' expr {$$ = opr('<', 2, $1, $3);}
80 | expr '>' expr {$$ = opr('>', 2, $1, $3);}
81 | expr LE expr {$$ = opr(LE, 2, $1, $3);}
82 | expr GE expr {$$ = opr(GE, 2, $1, $3);}
83 | expr EQ expr {$$ = opr(EQ, 2, $1, $3);}
84 | expr NE expr {$$ = opr(NE, 2, $1, $3);}
85 | '-' expr %prec UMINUS {$$ = opr(UMINUS, 1, $2);}
86 | '(' expr ')' {$$ = $2;}
87 ;
88
89 %%
90
91
92
93 node* con (int value)
94 {
95     node *p = (node*)malloc(sizeof(node));
96     if(p == NULL)
97         yyerror(Heap is full);
98
99     p->node_type = type_const;
100    p->cn.value = value;
101
102    return p;
103 }
104
105 node* id (int index)
106 {
107     node *p = (node*)malloc(sizeof(node));
108     if(p == NULL)
109         yyerror(Heap is full);
110
111     p->node_type = type_id;
112     p->in.index = index;
113
114     return p;
115 }
116
117 node* opr (int opr, int nop, ...)

```

```
118 {
119     va_list ap;
120     int i;
121
122     node *p = (node*)malloc(sizeof(node) + (nop-1) * sizeof(node*));
123     if(p == NULL)
124         yyerror(Heap is full);
125
126     p->node_type = type_op;
127     p->on.opr = opr;
128     p->on.nop = nop;
129
130     va_start(ap, nop);
131     for(i = 0; i < nop; i++)
132         p->on.op[i] = va_arg(ap, node*);
133
134     va_end(ap);
135     return p;
136 }
137
138 void free_node(node* p)
139 {
140     if(!p)
141         return;
142
143     if(p->node_type == type_op)
144     {
145         int i;
146         for(i = 0; i < p->on.nop; i++)
147             free_node(p->on.op[i]);
148     }
149
150     free(p);
151 }
152
153 void yyerror (char *s)
154 {
155     printf(int main())yyvsparse();return 0;
```

Listing 3: Implementación del Parser en YACC

4 Interpreter

4.1 ¿Qué es un interprete?

El intérprete analiza a través del árbol de sintaxis abstracto generado en el archivo del Parser, para obtener la salida. Lo hace mediante el uso de la recursión, ya que *ex()* se llama de forma recursiva mientras se calculan subárboles de un nodo y un caso de cambio, ya que hay diferentes casos para diferentes tipos de nodo, además de *opr* tenemos un caso de cambio anidado para todas las diferentes operaciones.

5 Diseño del sistema

En la Figura 3, se muestra cómo se construye el intérprete y las dependencias que existen entre los diferentes archivos.

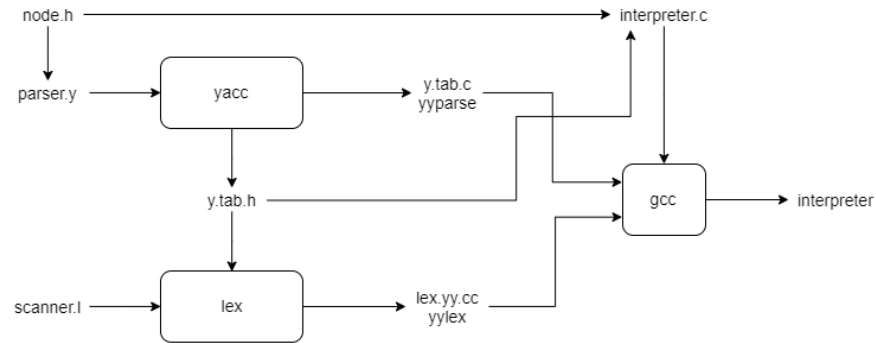


Figure 3: Construcción del intérprete

El siguiente script de shell se utilizará para la compilación del intérprete, como se muestra a continuación:

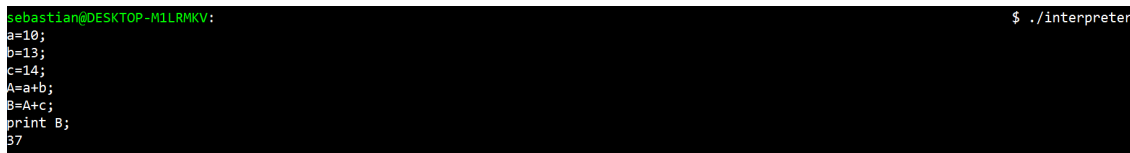
```
1  bison -y -d parser.y
2  flex scanner.l
3  gcc -c y.tab.c lex.yy.c
4  gcc y.tab.o lex.yy.o interpreter.c -o interpreter
```

6 Prueba del interprete

A continuación mostraremos algunas capturas de pantalla del interprete creado, con el fin de probar las diferentes funcionalidades.

6.1 Operaciones aritméticas

En la Figura 4 se muestra la ejecución de código que muestra las operaciones aritméticas, así como la forma en que siguen la asociatividad y la precedencia.

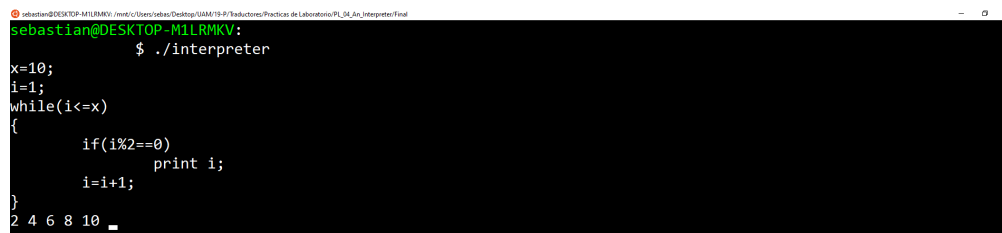


```
sebastian@DESKTOP-M1LRMKV: $ ./interpreter
a=10;
b=13;
c=14;
A=a+b;
B=A+c;
print B;
37
```

Figure 4: Prueba de operaciones aritméticas

6.2 Loops iterativos

En la Figura 5 se muestra la ejecución de código muestra declaraciones iterativas y condicionales, junto con la operación relacional.



```
sebastian@DESKTOP-M1LRMKV: $ ./interpreter
x=10;
i=1;
while(i<=x)
{
    if(i%2==0)
        print i;
    i=i+1;
}
2 4 6 8 10 _
```

Figure 5: Condiciones iterativas

References

- Niemann, T. (2004). A compact guide to lex & yacc.
<http://epaperpress.com/lexandyacc/download/lexyacc.pdf>.
- Rojas, S. G. and Mata, M. Á. M. (2005). *Java a Tope: Traductores Y Compiladores Con Lex/yacc, Jflex/cup Y Javacc*. Sergio Gálvez Rojas.