



Tecnológico de Monterrey

Compilador C-
Diseño de Compiladores
Profesor Victor De la Cueva
Luis Enrique Güitrón Leal
A01018616

Código Generado

El código que se genera con este compilador es código MIPS, se decidió utilizar este lenguaje ensamblador debido a la facilidad de emular un procesador MIPS con software libre. Por otro lado, se escogió este ensamblador debido a que fue el que se utilizó en el curso de Diseño de Compiladores en el semestre Enero – Mayo 2019.

Manual de Usuario

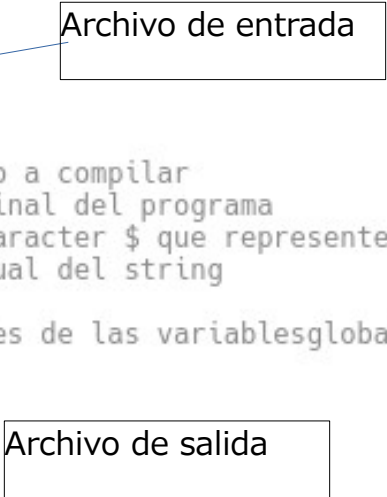
Para compilar un programa en C- se deben seguir los siguientes pasos:

- Descargar el código proporcionado en python 3 manteniendo todos los archivos en la misma carpeta.
- Escribir programa en C- (ver descripción del lenguaje e apéndice D).
- Modificar el archivo main.py para elegir el nombre del archivo de entrada (programa escrito en C-) así como el archivo de salida (programa escrito en MIPS).

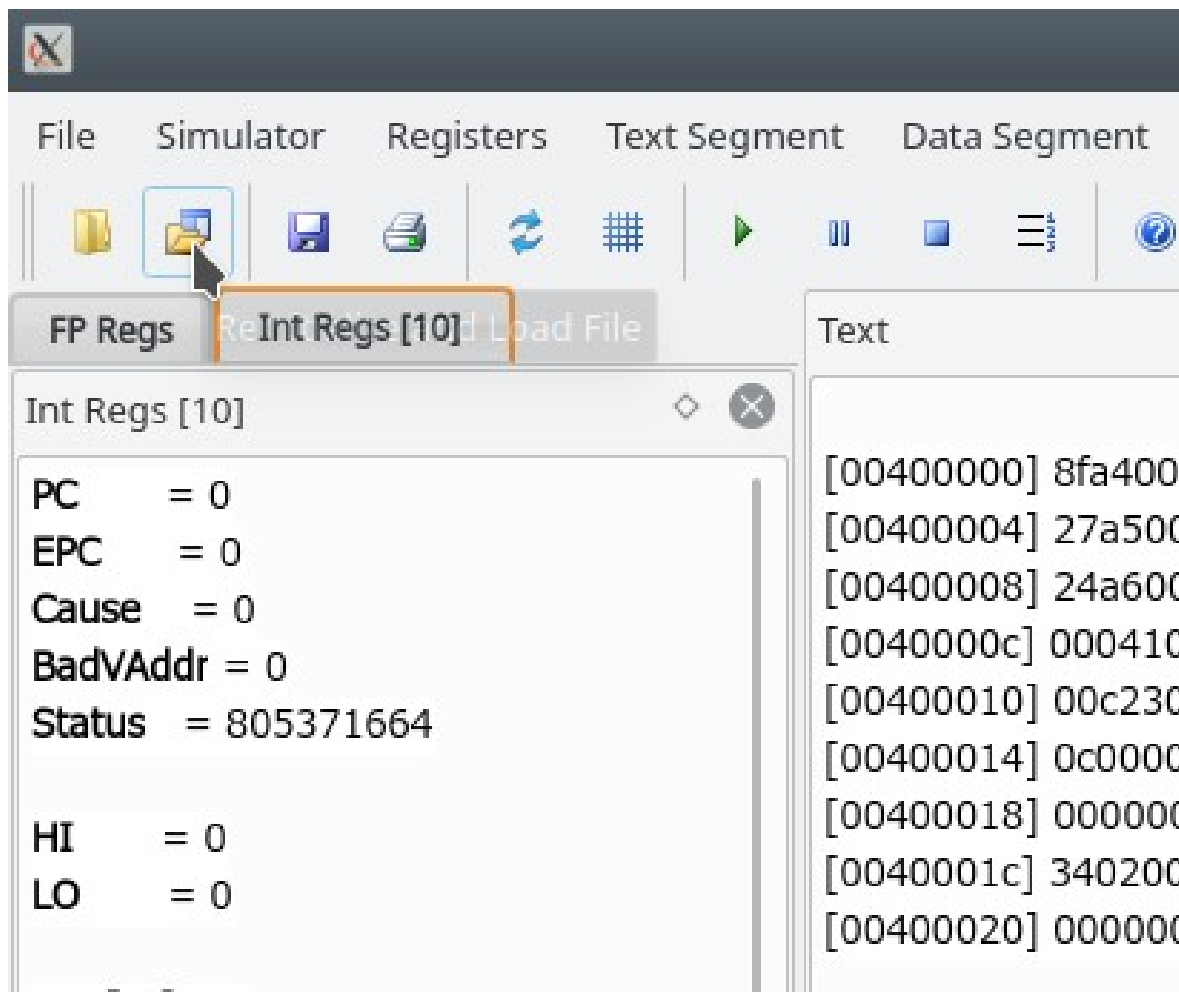
```
from globalTypes import *
from Parser import *
from semantica import *
from cgen import *

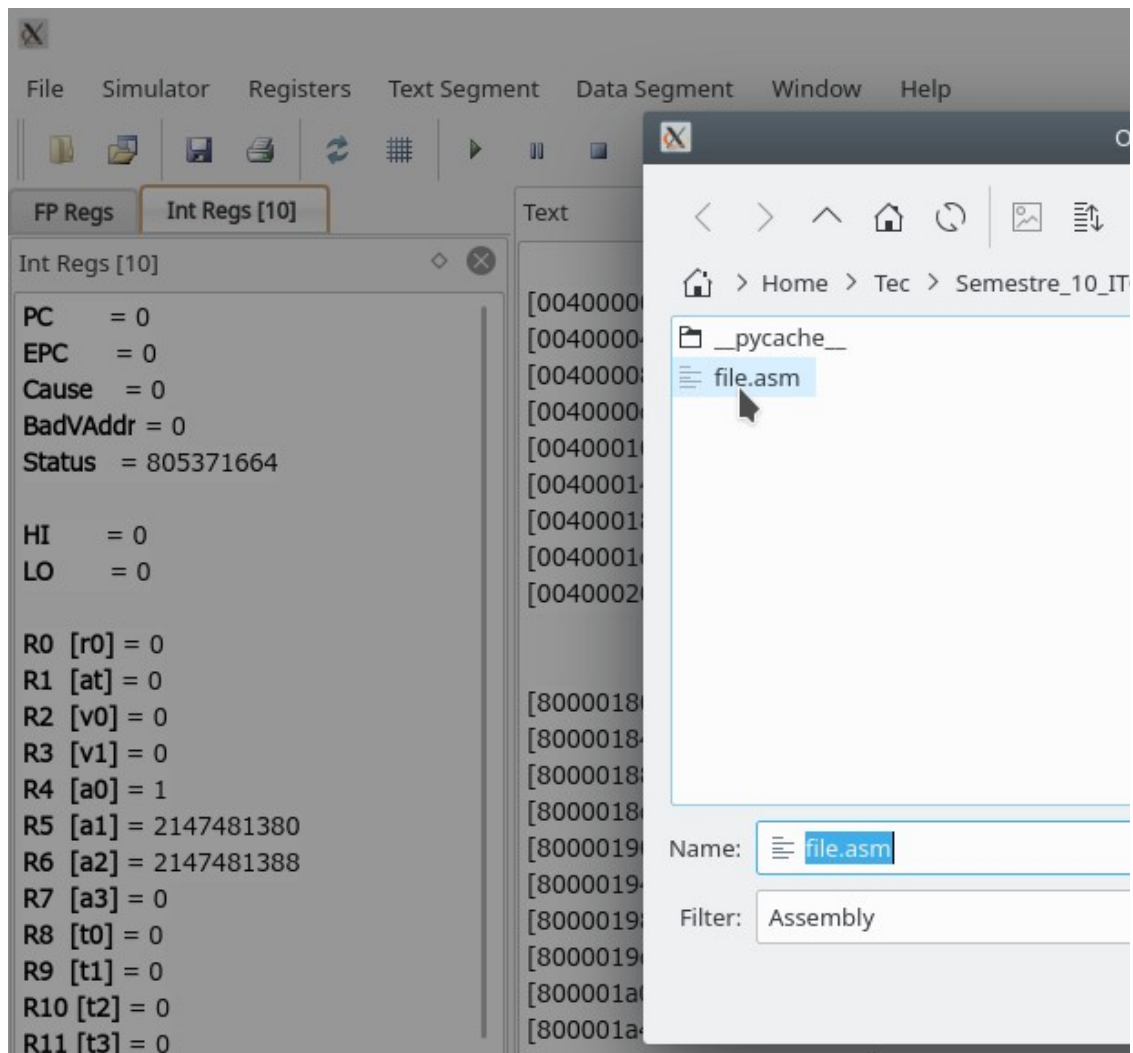
f = open('sample.c-', 'r')
programa = f.read() # lee todo el archivo a compilar
progLong = len(programa) # longitud original del programa
programa = programa + '$' # agregar un caracter $ que represente EOF
posicion = 0 # posición del caracter actual del string

# función para pasar los valores iniciales de las variables globales
globales(programa, posicion, progLong)
AST = parser(False)
semantica(AST, False)
codeGen(AST, "file.asm")
```

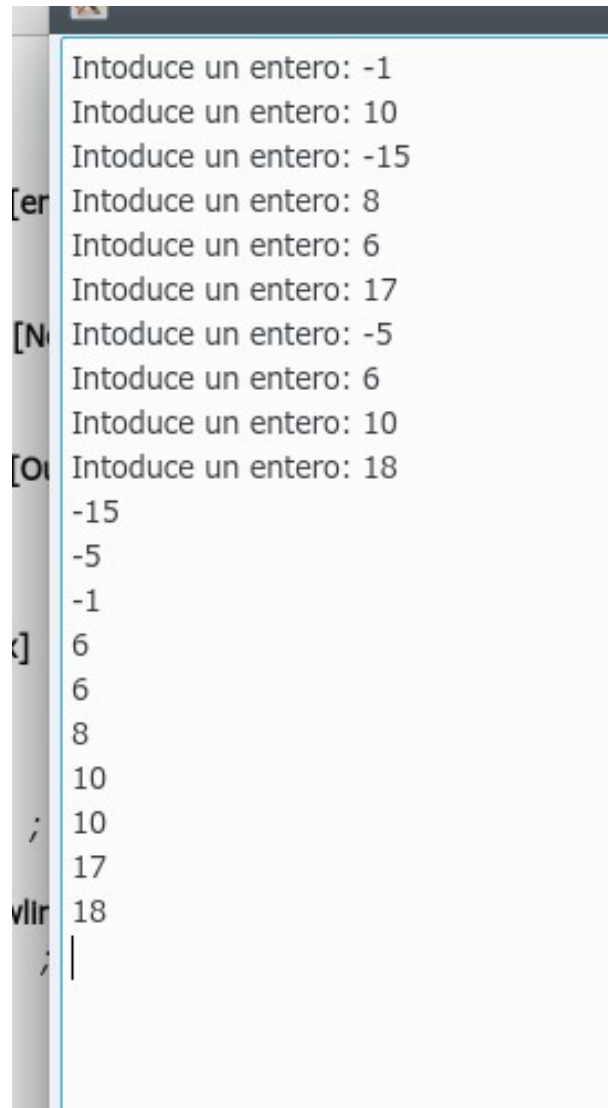


- Ejecutar el archivo main.py con el comando `python3 main.py`.
- Si hubo errores en la compilación se mostrará el error y no se generará el programa en MIPS.
- En caso de error se indicará el tipo de error (léxico, sintáctico o semántico) y su ubicación.
- Si el programa es compilado exitosamente se podrá probar la ejecución en el emulador de MIPS qtspim <http://spimsimulator.sourceforge.net/>.
- Dentro del emulador cargar el archivo generado por el compilador con terminación `.asm`





- El resultado de la ejecución se podrá ver en la consola de qtspim
- A continuación se muestra el ejemplo de un programa que hace ordenación de enteros



```

Intoduce un entero: -1
Intoduce un entero: 10
Intoduce un entero: -15
[er Intoduce un entero: 8
Intoduce un entero: 6
Intoduce un entero: 17
[N Intoduce un entero: -5
Intoduce un entero: 6
Intoduce un entero: 10
[O Intoduce un entero: 18
-15
-5
-1
6
6
8
10
; 10
17
vlin 18
; |
  
```

- En caso de utilizar entradas en la consola se debe introducir un entero y se debe presionar la tecla 'Enter' para continuar con la ejecución del programa.
- Los errores de Runtime (índice fuera de rango) se mostrarán en la consola de qtspim en caso de presentarse.

Apéndices

A continuación se incluyen los siguientes apéndices:

- A. Documentación Lexer
- B. Documentación Parser
- C. Documentación Analizador Semántico
- D. Definición de C-

Apéndice A Documentación Lexer

Expresiones Regulares

Token	Expresión Regular
ELSE	'else'
IF	'if'
INT	'int'
RETURN	'return'
VOID	'void'
WHILE	'while'
PLUS	'+'
MINUS	'-'
TIMES	'*'
DIVIDE	'/'
EQUALS	'='
LT	'<'
LE	'<='
GT	'>'
GE	'>='
EQ	'=='
NE	'!='
SEMICOLON	';'
COMMA	','
LPAREN	'('
RPAREN)'
LBRACKET	'['
RBRACKET	']'
LKEY	'{'

RKEY	'}'
ID	[a-zA-Z][a-zA-Z]*
NUM	[0-9][0-9]*
COMMENT	/'*(/* '*'* [~'*/])'*' '*'/'
ENDFILE	'\$'

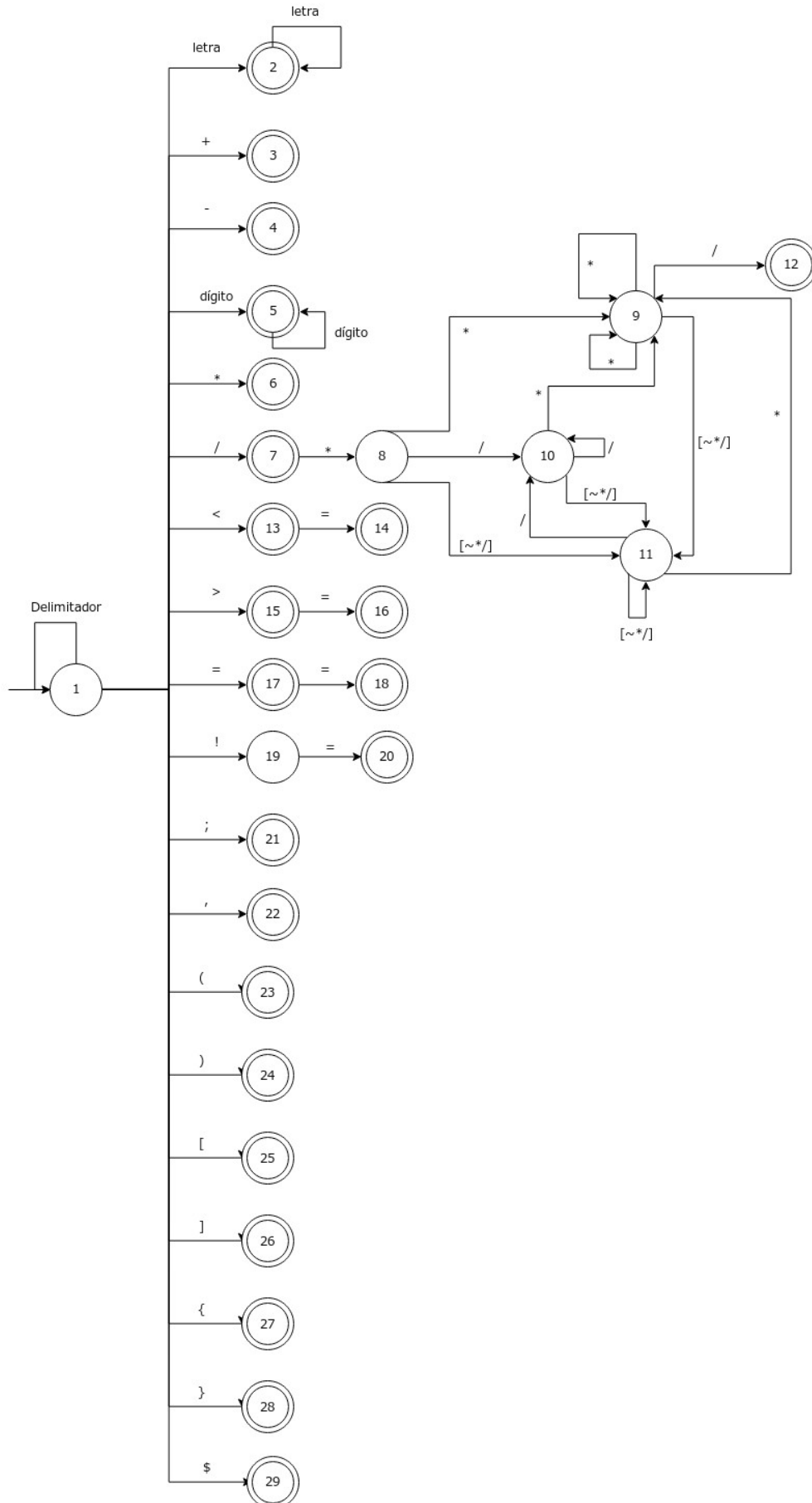
DFA

Tokens por estado final de DFA

Estado	Token
2	ID o palabra reservada (ELSE, IF, INT, RETURN, VOID, WHILE)
3	PLUS
4	MINUS
5	NUM
6	TIMES
7	DIVIDE
12	COMMENT
13	LT
14	LE
15	GT
16	GE
17	EQUALS
18	EQ
20	NE
21	SEMICOLON
22	COMMA

23	LPAREN
24	RPAREN
25	LBRACKET
26	RBRACKET
27	LKEY
28	RKEY
29	ENDFILE

DFA



Recuperación de Errores

En esta implementación el lexer revisa únicamente la cadena actual sin revisar los caracteres que vienen a después. Por la razón anterior, los errores identificados por el lexer corresponderán a caracteres invalidos.

Ejemplos:

Cadena	Salida Lexer	Notas
contador = contador + 3indice	(TokenType.ID, contador) (TokenType.EQUALS, =) (TokenType.ID, contador) (TokenType.PLUS, +) (TokenType.NUM, 3) (TokenType.ID, indice)	A pesar de que la cadena no sea válida en el lenguaje C-, el lexer identificará únicamente Tokens válidos, por lo que la identificación de este tipo de error será realizada por el parser.
variable = 1 + _3	(TokenType.ID, variable) (TokenType.EQUALS, =) (TokenType.NUM, 1) (TokenType.PLUS, +) (TokenType.ERROR, _) (TokenType.NUM, 3)	El caracter '_' no está definido, por lo que es detectado como un caracter ilegal por el lexer.

Al detectar un caracter inválido se imprimirá un mensaje de error indicando la línea y la posición de este caracter. El lexer continuará detectando tokens de forma normal inmediatamente después de la expresión inválida.

La impresión en consola de la línea

variable = 1 + _3

es la siguiente:

```
[lguitron24@Acer-Arch Proyect1Lexer]$ python main.py
( TokenType.ID , variable )
( TokenType.EQUALS , = )
( TokenType.NUM , 1 )
( TokenType.PLUS , + )
-----
( TokenType.ERROR , _ )
Linea 1 : ERROR, caracter inesperado
variable = 1 + _3
               ^
-----
( TokenType.NUM , 3 )
( TokenType.ENDFILE , $ )
[lguitron24@Acer-Arch Proyect1Lexer]$
```

Apéndice B Documentación Parser

Gramática en EBNF

1. $\text{program} \rightarrow \text{declaration-list}$
2. $\text{declaration-list} \rightarrow \text{declaration} \{ \text{declaration} \}$
3. $\text{declaration} \rightarrow \text{var-declaration} \mid \text{fun-declaration}$
4. $\text{var-declaration} \rightarrow \text{type-specifier ID} [\text{¥} [\text{NUM} \text{¥}]] ;$
5. $\text{type-specifier} \rightarrow \text{int} \mid \text{void}$
6. $\text{fun-declaration} \rightarrow \text{type-specifier ID} \text{¥} (\text{params} \text{¥}) \text{compound-stmt}$
7. $\text{params} \rightarrow \text{param-list} \mid \text{void}$
8. $\text{param-list} \rightarrow \text{param} \{ , \text{param} \}$
9. $\text{param} \rightarrow \text{type-specifier ID} [\text{¥} [\text{¥}]]$
10. $\text{compound-stmt} \rightarrow \text{¥} \{ \text{local-declarations statement-list} \text{¥} \}$
11. $\text{local-declarations} \rightarrow \{ \text{var_declaration} \}$
12. $\text{statement-list} \rightarrow \{ \text{statement} \}$
13. $\text{statement} \rightarrow \text{expression-stmt}$
 $\quad \mid \text{compound-stmt}$
 $\quad \mid \text{selection-stmt}$
 $\quad \mid \text{iteration-stmt}$
 $\quad \mid \text{return-stmt}$
14. $\text{expression-stmt} \rightarrow [\text{expression}] ;$
15. $\text{selection-stmt} \rightarrow \text{if} \text{¥} (\text{expression} \text{¥}) \text{compound_stmt} [\text{else compound_stmt}]$
16. $\text{iteration-stmt} \rightarrow \text{while} \text{¥} (\text{expression} \text{¥}) \text{compound_stmt}$
17. $\text{return-stmt} \rightarrow \text{return} [\text{expression}] ;$
18. $\text{expression} \rightarrow \{ \text{var} = \} \text{simple-expression}$
19. $\text{var} \rightarrow \text{ID} [\text{¥} [\text{expression} \text{¥}]]$
20. $\text{simple-expression} \rightarrow \text{additive-expression} [\text{relop additive-expression}]$
21. $\text{relop} \rightarrow < = \mid < \mid > \mid > = \mid == \mid !=$
22. $\text{additive-expression} \rightarrow \text{term} \{ \text{addop term} \}$
23. $\text{addop} \rightarrow + \mid -$
24. $\text{term} \rightarrow \text{factor} \{ \text{mulop factor} \}$
25. $\text{mulop} \rightarrow * \mid /$
26. $\text{factor} \rightarrow \text{NUM} \mid (\text{expression}) \mid \text{call} \mid \text{var}$
27. $\text{call} \rightarrow \text{ID} (\text{args})$
28. $\text{args} \rightarrow \text{arg-list} \mid \epsilon$
29. $\text{arg-list} \rightarrow \text{expression} \{ , \text{expression} \}$

Apéndice C Documentación Analizador Semántico

Reglas de inferencia de tipos

$$1. \frac{i \text{ es una literal entera}}{\vdash i:int}$$

$$2. \frac{\vdash e_1:int \quad \vdash e_2:int}{\vdash e_1 + e_2:int}$$

$$3. \frac{\vdash e_1:int \quad \vdash e_2:int}{\vdash e_1 - e_2:int}$$

$$4. \frac{\vdash e_1:int \quad \vdash e_2:int}{\vdash e_1 * e_2:int}$$

$$5. \frac{\vdash e_1:int \quad \vdash e_2:int}{\vdash e_1/e_2:int}$$

$$6. \frac{\vdash e_1:int \quad \vdash e_2:int}{\vdash e_1 < e_2:int}$$

$$7. \frac{\vdash e_1:int \quad \vdash e_2:int}{\vdash e_1 \leq e_2:int}$$

$$8. \frac{\vdash e_1:int \quad \vdash e_2:int}{\vdash e_1 > e_2:int}$$

$$9. \frac{\vdash e_1:int \quad \vdash e_2:int}{\vdash e_1 \geq e_2:int}$$

$$10. \frac{\vdash e_1:int \quad \vdash e_2:int}{\vdash e_1 == e_2:int}$$

$$11. \frac{\vdash e_1:int \quad \vdash e_2:int}{\vdash e_1! = e_2:int}$$

$$12. \frac{\vdash e_1:int \quad e_1 \text{ tiene parametros}}{\vdash e_1:int \text{ funcion}}$$

$$13. \frac{\vdash e_1:void \quad e_1 \text{ no tiene parametros}}{\vdash e_1:int \text{ variable}}$$

$$14. \frac{\vdash e_1:void \quad e_1 \text{ tiene parametros}}{\vdash e_1:void \text{ funcion}}$$

Nota: Para determinar si un nombre de tipo 'int' de la tabla de símbolos es función o variable se revisa el diccionario de sus propiedades, si tiene una entrada llamada 'params' se determina que es función, de lo contrario se determina que es variable.

Estructura de tabla de símbolos

Las tablas de símbolos se almacenan en estructura de árbol en donde la raíz del árbol representa el scope global, cada tabla de símbolos puede tener de 0 a varios hijos.

Se entra a un nuevo scope en C- al entrar a un *"compound statement"* (declaración encerrada por llaves `{}`).

Cada uno de los nombres introducidos en las tablas de símbolos se relaciona con dos propiedades:

1. Tipo : `int`, `int[]` o `void`

2. Diccionario: propiedades adicionales de la variable o función.

Ejemplos:

<code>{}</code>	- Diccionario vacío utilizado en <code>int</code>
<code>{'size' : 10}</code>	- Propiedad <code>size</code> utilizada en arreglos
<code>{'params': void}</code>	- Propiedad utilizada en funciones sin parámetros
<code>{'params': ['int[]', 'int']}</code>	- Propiedad utilizada en funciones con parámetros

A continuación se muestra el ejemplo de un programa en C- junto con su respectiva tabla de símbolos:

Programa

```
int var;
int arr[12];

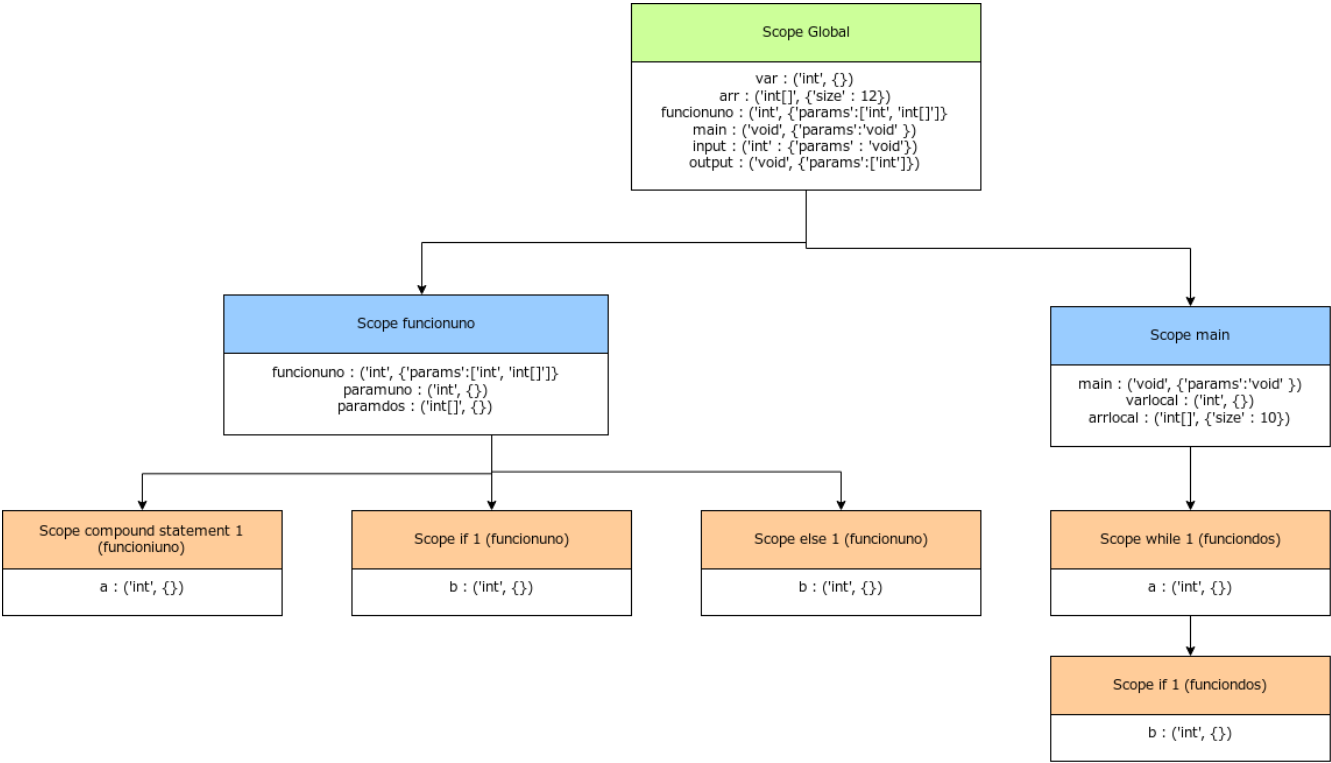
int funcionuno(int paramuno, int paramdos[])
{
    {
        int a;
    }

    if(paramuno)
    {
        int b;
    }
    else
    {
        int b;
    }
    return paramuno;
}

void main(void)
{
    int varlocal;
    int arrlocal[10];

    while(varlocal)
    {
        int a;
        if(a)
        {
            int b;
        }
    }
}
```


Tablas de símbolos



C- (C-minus)

Un lenguaje para un proyecto de compilador

(Lauden, 2004)

- Es un subconjunto considerablemente restringido de C.
- Contiene:
 - Enteros
 - Arreglos de enteros
 - Funciones (con tipo o **void**)
 - Declaraciones (estáticas) locales y globales
 - Funciones recursivas (simples)
 - Condicional **if-else**
 - Ciclo **while**
 - Función **input()** que lee desde el teclado
 - Función **output()** que escribe a la pantalla
- Un programa se compone de una secuencia de declaraciones de variables y funciones.
- Al final debe declararse una función **main**.
- La ejecución inicia con una llamada a **main**.

Léxico de C-

1. Las palabras clave o reservadas del lenguaje son las siguientes:

else if int return void while

Todas las palabras reservadas o clave están reservadas, y deben ser escritas en minúsculas.

2. Los símbolos especiales son los siguientes:

+ - * / < <= > >= == != = ; , () [] { } /* */

3. Otros tokens son **ID** y **NUM**, definidos mediante las siguientes expresiones regulares:

ID = letra letra*
NUM = dígito dígito*
letra = a|..|z|A|..|Z
dígito = 0|..|9

Se distingue entre letras minúsculas y mayúsculas.

4. Los espacios en blanco se componen de blancos, retornos de línea y tabulaciones. El espacio en blanco es ignorado, excepto cuando deba separar **ID**, **NUM** y palabras reservadas.
5. Los comentarios están encerrados entre las anotaciones habituales del lenguaje C **/* . . . */**. Los comentarios se pueden colocar en cualquier lugar donde pueda aparecer un espacio en blanco (es decir, los comentarios no pueden ser colocados dentro de los token) y pueden incluir más de una línea. Los comentarios no pueden estar anidados.

Sintaxis de C-

Una gramática BNF para C- es como se describe a continuación:

1. *program* → *declaration-list*
2. *declaration-list* → *declaration-list declaration* | *declaration*
3. *declaration* → *var-declaration* | *fun-declaration*
4. *var-declaration* → *type-specifier ID ;* | *type-specifier ID [NUM] ;*
5. *type-specifier* → **int** | **void**
6. *fun-declaration* → *type-specifier ID (params) compound-stmt*
7. *params* → *param-list* | **void**
8. *param-list* → *param-list , param* | *param*
9. *param* → *type-specifier ID* | *type-specifier ID []*
10. *compound-stmt* → { *local-declarations statement-list* }
11. *local-declarations* → *local-declarations var-declaration* | *empty*
12. *statement-list* → *statement-list statement* | *empty*
13. *statement* → *expression-stmt* | *compound-stmt* | *selection-stmt*
| *iteration-stmt* | *return-stmt*
14. *expression-stmt* → *expression ;* | *;*
15. *selection-stmt* → **if** (*expression*) *statement*
| **if** (*expression*) *statement* **else** *statement*
16. *iteration-stmt* → **while** (*expression*) *statement*
17. *return-stmt* → **return ;** | **return** *expression ;*
18. *expression* → *var = expression* | *simple-expression*
19. *var* → *ID* | *ID [expression]*
20. *simple-expression* → *additive-expression relop additive-expression*
| *additive-expression*
21. *relop* → **<=** | **<** | **>** | **>=** | **==** | **!=**
22. *additive-expression* → *additive-expression addop term* | *term*
23. *addop* → **+** | **-**
24. *term* → *term mulop factor* | *factor*
25. *mulop* → ***** | **/**
26. *factor* → (*expression*) | *var* | *call* | **NUM**
27. *call* → **ID** (*args*)
28. *args* → *arg-list* | *empty*
29. *arg-list* → *arg-list , expression* | *expression*

Semántica de C-

1. $program \rightarrow declaration-list$
2. $declaration-list \rightarrow declaration-list\ declaration \mid declaration$
3. $declaration \rightarrow var-declaration \mid fun-declaration$

Un programa (*program*) se compone de una lista (o secuencia) de declaraciones (*declaration-list*), las cuales pueden ser declaraciones de variable o función, en cualquier orden. Debe haber al menos una declaración. Las restricciones semánticas son como sigue (éstas no se presentan en C). Todas las variables y funciones deben ser declaradas antes de utilizarlas (esto evita las referencias de retroajuste). La última declaración en un programa debe ser una declaración de función con el nombre **main**. Advertir que C- carece de prototipos, de manera que no se hace una distinción entre declaraciones y definiciones (como en el lenguaje C).

4. $var-declaration \rightarrow type-specifier\ ID\ ; \mid type-specifier\ ID\ [\ NUM \]\ ;$
5. $type-specifier \rightarrow int \mid void$

Una declaración de variable declara una variable simple de tipo entero o una variable de arreglo cuyo tipo base es entero, y cuyos índices abarcan desde $0 \dots NUM-1$. Observe que en C- los únicos tipos básicos son entero y vacío ("void"). En una declaración de variable sólo se puede utilizar el especificador de tipo **int**. **void** es para declaraciones de función (véase más adelante). Advertir también que sólo se puede declarar una variable por cada declaración.

6. $fun-declaration \rightarrow type-specifier\ ID\ (\ params \)\ compound-stmt$
7. $params \rightarrow param-list \mid void$
8. $param-list \rightarrow param-list\ ,\ param \mid param$
9. $param \rightarrow type-specifier\ ID \mid type-specifier\ ID\ [\]$

Una declaración de función consta de un especificador de tipo (*type-specifier*) de retorno, un identificador y una lista de parámetros separados por comas dentro de paréntesis, seguida por una sentencia compuesta con el código para la función. Si el tipo de retorno de la función es **void**, entonces la función no devuelve valor alguno (es decir, es un procedimiento). Los parámetros de una función pueden ser **void** (es decir, sin parámetros) o una lista que representa los parámetros de la función. Los parámetros seguidos por corchetes son parámetros de arreglo cuyo tamaño puede variar. Los parámetros enteros simples son pasados por valor. Los parámetros de arreglo son pasados por referencia (es decir, como punteros) y deben ser igualados mediante una variable de arreglo durante una llamada. Advertir que no hay parámetros de tipo "función". Los parámetros de una función tienen un ámbito igual a la sentencia compuesta de la declaración de función, y cada invocación de una función tiene un conjunto separado de parámetros. Las funciones pueden ser recursivas (hasta el punto en que la declaración antes del uso lo permita).

10. $compound-stmt \rightarrow \{ \ local-declarations\ statement-list \}$

Una sentencia compuesta se compone de llaves que encierran un conjunto de declaraciones y sentencias. Una sentencia compuesta se realiza al ejecutar la secuencia de sentencias

en el orden dado. Las declaraciones locales tienen un ámbito igual al de la lista de sentencias de la sentencia compuesta y reemplazan cualquier declaración global.

11. *local-declarations* \rightarrow *local-declarations* *var-declaration* | *empty*

12. *statement-list* \rightarrow *statement-list* *statement* | *empty*

Advierta que tanto la lista de declaraciones como la lista de sentencias pueden estar vacías. (El no terminal *empty* representa la cadena vacía, que se describe en ocasiones como *ε*.)

13. *statement* \rightarrow *expression-stmt*
 | *compound-stmt*
 | *selection-stmt*
 | *iteration-stmt*
 | *return-stmt*

14. *expression-stmt* \rightarrow *expression* ; | ;

Una sentencia de expresión tiene una expresión opcional seguida por un signo de punto y coma. Tales expresiones por lo regular son evaluadas por sus efectos colaterales. Por consiguiente, esta sentencia se utiliza para asignaciones y llamadas de función.

15. *selection-stmt* \rightarrow **if** (*expression*) *statement*
 | **if** (*expression*) *statement* **else** *statement*

La sentencia **if** tiene la semántica habitual: la expresión es evaluada; un valor distinto de cero provoca la ejecución de la primera sentencia; un valor de cero ocasiona la ejecución de la segunda sentencia, si es que existe. Esta regla produce la ambigüedad clásica del **else** ambiguo, la cual se resuelve de la manera estándar: la parte **else** siempre se analiza sintácticamente de manera inmediata como una subestructura del **if** actual (la regla de eliminación de ambigüedad “de anidación más cercana”).

16. *iteration-stmt* \rightarrow **while** (*expression*) *statement*

La sentencia **while** es la única sentencia de iteración en el lenguaje C—. Se ejecuta al evaluar de manera repetida la expresión y al ejecutar entonces la sentencia si la expresión evalúa un valor distinto de cero, finalizando cuando la expresión se evalúa a 0.

17. *return-stmt* \rightarrow **return** ; | **return** *expression* ;

Una sentencia de retorno puede o no devolver un valor. Las funciones no declaradas como **void** deben devolver valores. Las funciones declaradas **void** no deben devolver valores. Un retorno provoca la transferencia del control de regreso al elemento que llama (o la terminación del programa si está dentro de **main**).

18. *expression* \rightarrow *var* = *expression* | *simple-expression*

19. *var* \rightarrow *ID* | *ID* [*expression*]

Una expresión es una referencia de variable seguida por un símbolo de asignación (signo de igualdad) y una expresión, o solamente una expresión simple. La asignación tiene la semántica de almacenamiento habitual: se encuentra la localidad de la variable representada por *var*, luego se evalúa la subexpresión a la derecha de la asignación, y se almacena el valor de la subexpresión en la localidad dada. Este valor también es devuelto como el valor de la expresión completa. Una *var* es una variable (entera) simple o bien una variable de arreglo subíndizada. Un subíndice negativo provoca que el programa se detenga (a diferencia de C). Sin embargo, no se verifican los límites superiores de los subíndices.

Las variables representan una restricción adicional en C^- respecto a C . En C el objetivo de una asignación debe ser un **valor l**, y los valores l son direcciones que pueden ser obtenidas mediante muchas operaciones. En C^- los únicos valores l son aquellos dados por la sintaxis de *var*, y así esta categoría es verificada sintácticamente, en vez de hacerlo durante la verificación de tipo como en C . Por consiguiente, en C^- está prohibida la aritmética de apuntadores.

20. $simple-expression \rightarrow additive-expression \text{ relop } additive-expression$
 $\quad \quad \quad | \quad additive-expression$

21. $relop \rightarrow <= \mid < \mid > \mid >= \mid == \mid !=$

Una expresión simple se compone de operadores relacionales que no se asocian (es decir, una expresión sin paréntesis puede tener solamente un operador relacional). El valor de una expresión simple es el valor de su expresión aditiva si no contiene operadores relacionales, o bien, 1 si el operador relacional se evalúa como verdadero, o 0 si se evalúa como falso.

22. $additive-expression \rightarrow additive-expression \text{ addop } term \mid term$

23. $addop \rightarrow + \mid -$

24. $term \rightarrow term \text{ mulop } factor \mid factor$

25. $mulop \rightarrow * \mid /$

Los términos y expresiones aditivas representan la asociatividad y precedencia típicas de los operadores aritméticos. El símbolo $/$ representa la división entera; es decir, cualquier residuo es truncado.

26. $factor \rightarrow (expression) \mid var \mid call \mid \mathbf{NUM}$

Un factor es una expresión encerrada entre paréntesis, una variable, que evalúa el valor de su variable; una llamada de una función, que evalúa el valor devuelto de la función; o un **NUM**, cuyo valor es calculado por el analizador léxico. Una variable de arreglo debe estar sub-indizada, excepto en el caso de una expresión compuesta por una **ID** simple y empleada en una llamada de función con un parámetro de arreglo (véase a continuación).

27. $call \rightarrow \mathbf{ID} \ (\ args \)$

28. $args \rightarrow arg-list \mid empty$

29. $arg-list \rightarrow arg-list , expression \mid expression$

Una llamada de función consta de un **ID** (el nombre de la función), seguido por sus argumentos encerrados entre paréntesis. Los argumentos pueden estar vacíos o estar compuestos por una lista de expresiones separadas mediante comas, que representan los valores que se asignarán a los parámetros durante una llamada. Las funciones deben ser declaradas antes de llamarlas, y el número de parámetros en una declaración debe ser igual al número de argumentos en una llamada. Un parámetro de arreglo en una declaración de función debe coincidir con una expresión compuesta de un identificador simple que representa una variable de arreglo.

Finalmente, las reglas anteriores no proporcionan sentencia de entrada o salida. Debemos incluir tales funciones en la definición de C++, puesto que a diferencia del lenguaje C, C++ no tiene facilidades de ligado o compilación por separado. Por lo tanto, consideraremos dos funciones por ser **predefinidas** en el ambiente global, como si tuvieran las declaraciones indicadas:

```
int input(void) { . . . }  
void output(int x) { . . . }
```

La función **input** no tiene parámetros y devuelve un valor entero desde el dispositivo de entrada estándar (por lo regular el teclado). La función **output** toma un parámetro entero, cuyo valor imprime a la salida estándar (por lo regular la pantalla), junto con un retorno de línea.

Ejemplos de programas en C-

El siguiente es un programa que introduce dos enteros, calcula su máximo común divisor y lo imprime:

```
/* Un programa para realizar el algoritmo
   de Euclides para calcular mcd. */

int gcd (int u, int v)
{ if (v == 0) return u ;
  else return gcd(v,u-u/v*v);
  /* u-u/v*v == u mod v */
}

void main(void)
{ int x; int y;
  x = input(); y = input();
  output(gcd(x,y));
}
```

A continuación tenemos un programa que introduce una lista de 10 enteros, los clasifica por orden de selección, y los exhibe otra vez:

```
/* Un programa para realizar ordenación por
   selección en un arreglo de 10 elementos. */

int x[10];

int minloc ( int a[], int low, int high )
{ int i; int x; int k;
  k = low;
  x = a[low];
  i = low + 1;
  while (i < high)
  { if (a[i] < x)
    { x = a[i];
      k = i; }
    i = i + 1;
  }
  return k;
}

void sort( int a[], int low, int high)
```

```

{ int i; int k;
  i = low;
  while (i < high-1)
    { int t;
      k = minloc(a,i,high);
      t = a[k];
      a[k] = a[i];
      a[i] = t;
      i = i + 1;
    }
}

```

```

void main(void)
{ int i;
  i = 0;
  while (i < 10)
    { x[i] = input();
      i = i + 1; }
  sort(x,0,10);
  i = 0;
  while (i < 10)
    { output(x[i]);
      i = i + 1; }
}

```