

TECNOLÓGICO DE MONTERREY EN TOLUCA

Semestre Agosto-Diciembre 2019

DISEÑO DE COMPILADORES REPORTE

- Analizador Léxico
- Analizador Sintáctico
- Analizador Semántico
- Generación de Código

Elaborado por:

A01366101 **María Fernanda Delgado Radillo**



El propósito de este documento es reportar el proyecto realizado para la clase de Diseño de Compiladores del Semestre Agosto-Diciembre 2019 en el ITESM Campus Toluca.

INTRODUCCIÓN

*“Un **compilador** es un programa informático que traduce un programa escrito en un lenguaje de programación a otro lenguaje de programación. Usualmente el segundo lenguaje es lenguaje de máquina, pero también puede ser un código intermedio (bytecode), o simplemente texto. Este proceso de traducción se conoce como **compilación**.” (Ictea,2019).*

Durante el proceso de creación de un compilador existen diversas fases como las vistas en clase basándonos en la bibliografía asignada.

Para cada fase del compilador, como el análisis léxico, el análisis sintáctico, el análisis semántico y la generación de código se realizó una parte del proyecto utilizando el lenguaje de programación Python para su codificación. El propósito del compilador sería aceptar un archivo con código en el lenguaje C- para el que se proporcionó la estructura (anexa). Procesarlo a través de las distintas etapas para generar así código en lenguaje ensamblador que podría ser probado por el simulador spim para el procesador MIPS, el cual fue elegido por el docente para su prueba. Dicho simulador del procesador MIPS es comúnmente usado para proyectos de simulación en materia de electrónica y para el aprendizaje y práctica del lenguaje ensamblador.

“Spim es un simulador autónomo que ejecuta programas MIPS32. Lee y ejecuta programas en lenguaje ensamblador escritos para este procesador. Spim también proporciona un depurador simple y un conjunto mínimo de servicios del sistema operativo. Spim no ejecuta programas binarios (compilados)”. (SPIM MIPS Simulator-SourceForge.net)

Todas las etapas del compilador fueron desarrolladas en el lenguaje de programación Python y probadas con el IDE Spyder para python de la distribución para Python 3 de Anaconda Navigator, pues contiene los módulos importados para el funcionamiento del proyecto sin descargas adicionales.

Se utilizó la versión para Python 3, disponible en

<https://www.anaconda.com/>

Una vez instalado, en la pantalla de inicio se deberá seleccionar Spyder y abrir los archivos en el IDE.

PRIMERA ETAPA DEL COMPILADOR

ANALIZADOR LÉXICO

Utilizando herramientas como PLY para Python que simula la herramienta flex para el lenguaje c, se definieron los tokens del lenguaje basados en la estructura de C- con los siguientes resultados. Detección de errores básicos en variables y definición de tokens.

```
In [210]: runfile('C:/Users/MFDeIgado/Desktop/compilers-master/lexer/main.py',
wdir='C:/Users/MFDeIgado/Desktop/compilers-master/lexer')
VOID = void
ID = main
LPAREN = (
INT = int
ID = argc
RPAREN = )
LBLOCK = {
INT = int
ID = a
COMMA = ,
INT = int
ID = b
SEMICOLON = ;
ID = a
EQUAL = =
Syntax Error @ line 6:10
  a = 10r;
    ^
```

SEGUNDA ETAPA DEL COMPILADOR

ANALIZADOR SINTÁCTICO

En esta etapa se definieron las principales estructuras conocidas presentes en un programa siguiendo la estructura del lenguaje c-, siendo capaz de generar un árbol sintáctico como resultado del procesamiento.

Y la detección de errores de sintaxis.

```
In [206]: runfile('C:/Users/MFDeIgado/Music/PARSER_A01366101/testing.py', wdir='C:/
Users/MFDeIgado/Music/PARSER_A01366101')
Reloaded modules: lexer, nomatter
Syntax Error @ line 17:23
  output( gcd(x,y) );~
    ^

start program
  declaration list
  declaration list
  declaration list
  declaration
  fun declaration
  type specifier
  int
  gcd
  (
    params
    param list
    param list
    param
    type specifier
    int
  )
  compound stmt
  {
    statement list
```

TERCERA ETAPA DEL COMPILADOR

ANALIZADOR SEMÁNTICO

Durante esta etapa del compilador, se definieron reglas de inferencia y niveles de alcance para las tablas de símbolos que fueron presentadas en estructura de lista por cada alcance de recorrido. Obteniendo los siguientes resultados.

AST realizado por el parser de la segunda etapa.

```
In [196]: runfile('C:/Users/MFDeIgado/Desktop/compilers-master/semantic/testing.py',
wdir='C:/Users/MFDeIgado/Desktop/compilers-master/semantic')
start program
  declaration list
  declaration list
  declaration list
  declaration
    fun declaration
      type specifier
      int
gcd
(
  params
  param list
  param list
  param
    type specifier
    int
  u
)
compound stmt
{
  statement list
```

Terminal de IPython

Historial de comandos

Ejemplo del manejo de error semántico e impresión de tabla por alcance.

```
Error:
No se Encuentra en la tabla "input"

Tables
[{'alcance': 0, 'gcd': 'int, fun, int, u', 'main': 'void, fun, void'}, {'alcance': 1,
'u': 'int'}, {'alcance': 2, 'x': 'int', 'y': 'int'}]
An exception has occurred, use %tb to see the full traceback.
```

SystemExit

CUARTA ETAPA DEL COMPILADOR

GENERACIÓN DE CÓDIGO

Tomando en cuenta los ejemplos existentes de QTSpim los diferentes registros a utilizar y los comandos de las principales llamadas a ejecutar fueron tomadas en cuenta.

Data	Text
Text	
User Text Segment [00400000]..[00440000]	
[00400000] 8fa40000	lw \$4, 0(\$29) ; 183: lw \$a0 0(\$sp) # argc
[00400004] 27a50004	addiu \$5, \$29, 4 ; 184: addiu \$a1 \$sp 4 # argv
[00400008] 24ae0004	addiu \$6, \$5, 4 ; 185: addiu \$a2 \$a1 4 # envp
[0040000c] 00041080	sll \$2, \$4, 2 ; 186: sll \$v0 \$a0 2
[00400010] 00c23021	addu \$6, \$6, \$2 ; 187: addu \$a2 \$a2 \$v0
[00400014] 0c000000	jal 0x00000000 [main] ; 188: jal main
[00400018] 00000000	nop ; 189: nop
[0040001c] 3402000a	ori \$2, \$0, 10 ; 191: li \$v0 10
[00400020] 0000000c	syscall ; 192: syscall # syscall 10 (exit)
Kernel Text Segment [80000000]..[80010000]	
[80000180] 0001d821	addu \$27, \$0, \$1 ; 90: move \$k1 \$at # Save \$at
[80000184] 3c019000	lui \$1, -28672 ; 92: sw \$v0 \$1 # Not re-entrant and we can't
trust \$sp	
[80000188] ac220200	sw \$2, 512(\$1) ; 93: sw \$a0 \$2 # But we need to use these
[8000018c] 3c019000	lui \$1, -28672 ; 93: sw \$a0 \$2 # But we need to use these
registers	
[80000190] ac240204	sw \$4, 516(\$1)
[80000194] 401ae800	mfc0 \$26, \$13 ; 95: mfc0 \$k0 \$13 # Cause register
[80000198] 001a2082	srl \$4, \$26, 2 ; 96: srl \$a0 \$k0 2 # Extract ExCoDe Field
[8000019c] 3084001f	andi \$4, \$4, 31 ; 97: andi \$a0 \$a0 0x1f
[800001a0] 34020004	ori \$2, \$0, 4 ; 101: li \$v0 4 # syscall 4 (print_str)
[800001a4] 3c049000	lui \$4, -28672 [__mi_] ; 102: la \$a0 __mi_
[800001a8] 0000000c	syscall ; 103: syscall
[800001ac] 34020001	ori \$2, \$0, 1 ; 105: li \$v0 1 # syscall 1 (print_int)
[800001b0] 001a2082	srl \$4, \$26, 2 ; 106: srl \$a0 \$k0 2 # Extract ExCoDe Field
[800001b4] 3084001f	andi \$4, \$4, 31 ; 107: andi \$a0 \$a0 0x1f
[800001b8] 0000000c	syscall ; 108: syscall
[800001bc] 34020004	ori \$2, \$0, 4 ; 110: li \$v0 4 # syscall 4 (print_str)
[800001c0] 3344003c	andi \$4, \$26, 60 ; 111: andi \$a0 \$k0 0x3c
[800001c4] 3c019000	lui \$1, -28672 ; 112: lw \$a0 __excp(\$a0)
[800001c8] 00240821	addu \$1, \$1, \$4
[800001cc] 8c240180	lw \$4, 384(\$1)
[800001d0] 00000000	nop ; 113: nop

Desafortunadamente los resultados para esta etapa no fueron satisfactorios pues no reconocía algunos símbolos, por lo que añadí utf encoding, logrando modificar algunas líneas pero al añadir el archivo al QTSpim sale también que no reconoce o existe algún carácter inválido por lo que no logré comprobar la eficacia ni el funcionamiento de la generación de código mediante la escritura en el s. file

Reqs	Int Regs [16]	Data	Text
nt Regs [16]			
%C = 400014		User data segment [10000000]..[10040000]	
%PC = 0		[10000000]..[1003ffff] 00000000	
%ause = 0			
%adVAddr = 0			
%status = 3000fff0		User Stack [7ffff4c4]..[80000000]	
%I = 0		[7ffff4c4] 00000001 7ffff593 00000000	
%O = 0		[7ffff4d0] 7ffff5e1 7ffff5b6 7ffff5f5 7ffff543	
%R0 = 0		[7ffff4e0] 7ffff5f2 7ffff5f5 7ffff5d1 7ffff59f	
%R1 = 0		[7ffff4f0] 7ffff5e6 7ffff546 7ffff5db 7ffff5de	
%R2 = 0		[7ffff500] 7ffff5d4 7ffff5d7 7ffff5d6 7ffff5d2	
%R3 = 0		[7ffff510] 7ffff5d4 7ffff5b6 7ffff5c9 7ffff57a	
%R4 = 0		[7ffff520] 7ffff5c6 7ffff52a 7ffff5ec 7ffff5d1	
%R5 = 0		[7ffff530] 7ffff5b4 7ffff5b6 7ffff589 7ffff5d1	
%R6 = 1		[7ffff540] 7ffff526 7ffff502 7ffff5d9 7ffff5bb	
%R7 = 0		[7ffff550] 7ffff570 7ffff573 7ffff526 7ffff512	
%R8 = 0		[7ffff560] 7ffff573 7ffff5ed 7ffff5c3 7ffff59a	
%R9 = 0		[7ffff570] 7ffff57f 7ffff555 7ffff542 7ffff523	
%R10 = 0		[7ffff580] 7ffff5e9 7ffff5d7 00000000 00000000	
%R11 = 0		[7ffff590] 43000000 73552f3a 2f737265 6544464d	
%R12 = 0		[7ffff5a0] 6461676c 6442f6f 656d7563 2d73746e	
%R13 = 0		[7ffff5b0] 706d6f63 72656c69 616d2d73 72657473	
%R14 = 0		[7ffff5c0] 646f632f 6e654765 74617265 2f6e6f69	
%R15 = 0		[7ffff5d0] 656c6966 7700732e 69646e69 3a433d72	
%R16 = 0		[7ffff5e0] 4e49575c 53574f44 4f425600 534d5f58	
%R17 = 0		[7ffff5f0] 4e495749 4c415453 41505f4c 433d4854	
%R18 = 0		[7ffff600] 72505c3a 6172676f 6946206d 5c73656c	
%R19 = 0		[7ffff610] 6361724f 565c656c 75747269 6f426c61	
%R20 = 0		[7ffff620] 55005c78 50524553 49464f52 433d454c	
%R21 = 0		[7ffff630] 73555c3a 5c737265 6544464d 6461676c	
%R22 = 0		[7ffff640] 5355006f 41465245 4d3d454d 6c654446	
%R23 = 0		[7ffff650] 6f646167 45535500 4d6f4452 5f4e4941	
%R24 = 0		[7ffff660] 4d414f52 50474e49 49464f52 443d454c	

El output en spider es prácticamente el mismo de la parte 3 imprimiendo la tabla y el AST

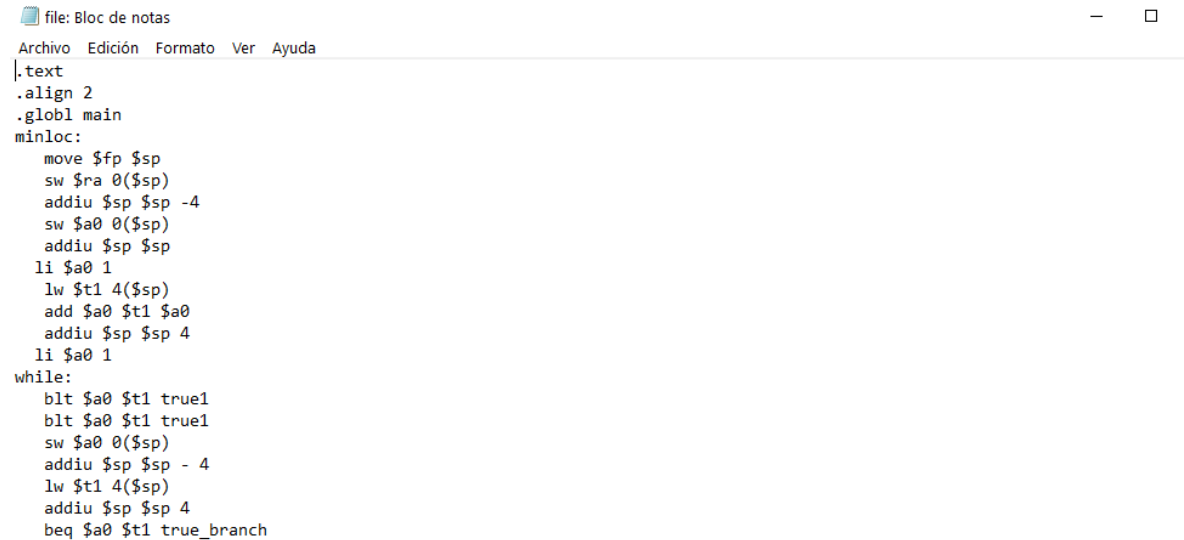
```

=
expression
  simple expression
  additive expression
  additive expression
  term
  factor
  var
  i
  addop
  +
  term
  factor
;
}
}
declaration
$
{'scope': 0, 'minloc': 'int,fun,int,a,int,low,int,high', 'sort':
'void,fun,int,a,int,low,int,high', 'main': 'void,fun,void'}
{'scope': 1, 'a': 'int', 'low': 'int', 'high': 'int', 'i': 'int', 'x': 'int', 'k':
'int'}
{'scope': 2, 'a': 'int', 'low': 'int', 'high': 'int', 'i': 'int', 'k': 'int', 't':
'int'}
{'scope': 3, 'i': 'int'}

In [40]:

```

Tras añadir el utf encoding logra escribir, pero no es reconocido a través del simulador QtSpim, por lo que no se puede comprobar que lo escrito en el documento sea congruente con la estructura de un programa en lenguaje ensamblador.



```
file: Bloc de notas
Archivo Edición Formato Ver Ayuda
|.text
.align 2
.globl main
minloc:
    move $fp $sp
    sw $ra 0($sp)
    addiu $sp $sp -4
    sw $a0 0($sp)
    addiu $sp $sp
    li $a0 1
    lw $t1 4($sp)
    add $a0 $t1 $a0
    addiu $sp $sp 4
    li $a0 1
while:
    blt $a0 $t1 true1
    blt $a0 $t1 true1
    sw $a0 0($sp)
    addiu $sp $sp - 4
    lw $t1 4($sp)
    addiu $sp $sp 4
    beq $a0 $t1 true_branch
```

Conclusión:

Este proyecto representó un reto intelectual de suma importancia para mí, pues tengo poco tiempo familiarizada con el lenguaje de programación Python, pero considero que puse a prueba mis capacidades y aunque no se logró la última parte del proyecto, las anteriores fueron satisfactorias, mismas que no imaginé pudiera realizar con los conocimientos que poseía en Python en ese momento.

De igual manera me brindó un panorama bastante claro del actuar detrás de un compilador pues cuando se tiene ya instalado, los procesos background son lo de menos.

DOCUMENTOS ANEXOS

DEFINICION DEL LENGUAJE C

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. $program \rightarrow declaration\text{-}list$
2. $declaration\text{-}list \rightarrow declaration\text{-}list\ declaration \mid declaration$
3. $declaration \rightarrow var\text{-}declaration \mid fun\text{-}declaration$

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

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

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 apuntadores) 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. $compound\text{-}stmt \rightarrow \{ local\text{-}declarations\ statement\text{-}list \}$

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. $local-declarations \rightarrow local-declarations \text{ var-declaration } | \text{ empty}$
12. $statement-list \rightarrow statement-list \text{ statement } | \text{ empty}$

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. $statement \rightarrow expression-stmt$
 $| \text{ compound-stmt}$
 $| \text{ selection-stmt}$
 $| \text{ iteration-stmt}$
 $| \text{ return-stmt}$
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. $selection-stmt \rightarrow \text{if } (\text{expression}) \text{ statement}$
 $| \text{if } (\text{expression}) \text{ statement else statement}$

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. $iteration-stmt \rightarrow \text{while } (\text{expression}) \text{ statement}$

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. $return-stmt \rightarrow \text{return } ; \text{ } | \text{return expression ;}$

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. $expression \rightarrow \text{var} = \text{expression} \text{ } | \text{ simple-expression}$
19. $\text{var} \rightarrow \text{ID} \text{ } | \text{ ID } [\text{expression}]$

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

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. $call \rightarrow ID \{ \text{ args } \}$
 28. $args \rightarrow arg-list \mid empty$
 29. $arg-list \rightarrow arg-list , \text{ expression } \mid \text{ expression}$

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.

LEXER

Regular Expressions defined

LBLOCK	= '{'	NUM	= '\d**'
RBLOCK	= '}'	LESSER.	= '<'
SEMICOLON.	= ';'.	GREATER.	= '>'
LPAREN.	= '('	MINUS.	= '-'
RPAREN.	= ')'	ENDFILE.	= '\\$'
PLUS.	= '+'	COMMA	= ','
EQUAL	= '='	TIMES	= '**'
COMPARE.	= '=='	DIVIDE	= '/'
LESSEREQ.	= '<='	LBRACKET.	= '['
NOTEQ.	= '>='	RBRACKET.	= ']'
GREATEREQ.	= '!='	COMMENT	= '\^*(\^(?!\\/)[\^*])\^*\^**'
IF	= 'if'	ID	= '[a-zA-Z][a-zA-Z0-9]**'
ELSE.	= 'else'	RESERVED.	= 'if else int void return'
WHILE.	= 'while'		
INT	= 'int'		
VOID	= 'void'		

PARSER

LEXER REGEX AND PARSER GRAMMATICAL RULES

```

LBLOCK      = '{'
RBLOCK      = '}'
SEMICOLON   = ';'
LPAREN      = '('
RPAREN      = ')'
PLUS        = '+'
EQUAL       = '='
COMPARE     = '<='
LESSEREQ    = '<='
NOTEQ       = '!='
GREATEREQ   = '>='
IF          = 'if'
ELSE        = 'else'
WHILE       = 'while'
INT         = 'int'
VOID        = 'void'
NUM = 'd'
LESSER      = '<'
GREATER     = '>'
MINUS       = '-'
ENDFILE     = '$'
COMMA       = ','
TIMES       = '*'
DIVIDE      = '/'
LBRACKET    = '['
RBRACKET    = ']'
COMMENT     = '/*(?!/*|/)/[*/]*/'
ID = '[a-zA-Z][a-zA-Z0-9]*'
RESERVED    = 'if|else|int|void|return'

```

```

X->Start
Start->Declarations
Declarations->fun_declara | var_declara
Declarations->EOF
Fun_declara | var_declara->type
Type->spec ID : | spec ID {NUM}
Spec->void | int
:
Fun_declara->spec(parameter) multstatement
Parameter->parameter list
Parameter list->parameter parameter
Multstatement->statement_list
Statement_list->statemnt,statemnt
Statemnt->expression|compound|selection|return
Expression->exp : ; | IF (exp) | while(exp) | var=exp 1exp operator factor
Return->return exp ;
Operator->GREATER,NE,EQ,COMPARE,LESS,PLUS,MINUS,TIMES,DIVIDE
Factor ->(exp) | ID | NUM | CALL
Call->ID(args)
Args->VOID, list
List->ag,arg

```

A01366101

Ma. Fernanda Delgado Radillo

SEMANTIC

SEMANTIC ANALYZER INFERENCE RULES AND SCOPE ORDER FOR SYMB TABLE LIST STRUCTURES

```

S' -> program
program -> declaration_list
declaration_list -> declaration_list declaration
declaration_list -> declaration
declaration -> var_declaration
declaration -> fun_declaration
var_declaration -> type_specifier ID
SEMICOLON
var_declaration -> type_specifier ID LBRACK
NUM RBRACK SEMICOLON type_specifier ->
INT
type_specifier -> VOID
fun_declaration -> type_specifier ID LPAREN
params RPAREN compound_stmt params ->
param_list
params -> VOID
param_list -> param_list COMMA param
param_list -> param
param -> type_specifier ID
param -> type_specifier LBRACK RBRACK
compound_stmt -> LCURLY local_declarations
statement_list RCURLY local_declarations ->
local_declarations var_declaration
local_declarations -> <empty>
statement_list -> statement_list statement
statement_list -> <empty>
statement -> expression_stmt
statement -> compound_stmt
statement -> selection_stmt
statement -> iteration_stmt
statement -> return_stmt
    
```

```

expression_stmt -> expression SEMICOLON
expression_stmt -> SEMICOLON
selection_stmt -> IF LPAREN expression RPAREN
statement
selection_stmt -> IF LPAREN expression RPAREN
statement ELSE statement iteration_stmt ->
WHILE LPAREN expression RPAREN statement
return_stmt -> RETURN SEMICOLON
return_stmt -> RETURN expression SEMICOLON
expression -> var EQUALS expression
expression -> simple_expression
var -> ID
var -> ID LBRACK expression RBRACK
simple_expression -> additive_expression relop
additive_expression simple_expression ->
additive_expression
relop -> LTHANEQ
relop -> LTHAN
relop -> GTHAN
relop -> GTHANEQ
relop -> EQUALTO
relop -> NOTEQUALTO
additive_expression -> additive_expression addop
term
additive_expression -> term
addop -> PLUS
addop -> MINUS
term -> term mulop factor
term -> factor
mulop -> TIMES
mulop -> DIVIDE
factor -> LPAREN expression RPAREN
factor -> ID
factor -> NUM
    
```

Voids	2
Declars	1
Expressions Vars Stmt List Compound_Stmt Stmts... Local_Declarations Arg List	0

A01366101