TECNOLÓGICO DE MONTERREY®

Project Simphony

Ali Ghahraei Figueroa
A01321877

Jorge Luis Márquez Sánchez
A01139543

03 de mayo de 2017

1. DESCRIPCIÓN Y DOCUMENTACIÓN DEL PROYECTO

1.1. DESCRIPCIÓN DEL PROYECTO

1.1.1. Visión, Objetivos y Alcance del Proyecto

- Visión

Nuestra visión a futuro de nuestro proyecto es que México cuente con los recursos necesarios para impulsar la enseñanza o el autoaprendizaje de la programación. Nuestro proyecto fue pensado para ser utilizado por personas que no necesariamente tengan nociones por la programación, pero que sí busquen darle una oportunidad de aprender sus conceptos básicos, además de que esta herramienta podrá ser adaptada fácilmente en ambientes escolares.

- Objetivos

Simphony tiene el objetivo de ser un lenguaje sencillo, en inglés y rápido de aprender para que cualquier persona pueda aprender la lógica básica de la programación, brindándoles los conocimientos necesarios para que por sí mismos exploren otros lenguajes más maduros como Python. La característica más notable de Simphony es que permite a los amantes de la música aprender de una forma más intuitiva, ya que nuestro lenguaje está diseñado para está audiencia en específico.

- Alcance del Proyecto

Simphony es un lenguaje programable en nuestra aplicación web, que se compila mediante una petición a nuestro servidor, la cual envía una respuesta según el código generado por el usuario. Este lenguaje cuenta con los elementos comunes de la mayoría de los lenguajes modernos, como lo son los arreglos, ciclos, condicionales, variables, funciones, funciones especiales, entre otros.

1.1.2. Análisis de Requerimientos y Casos de Uso generales

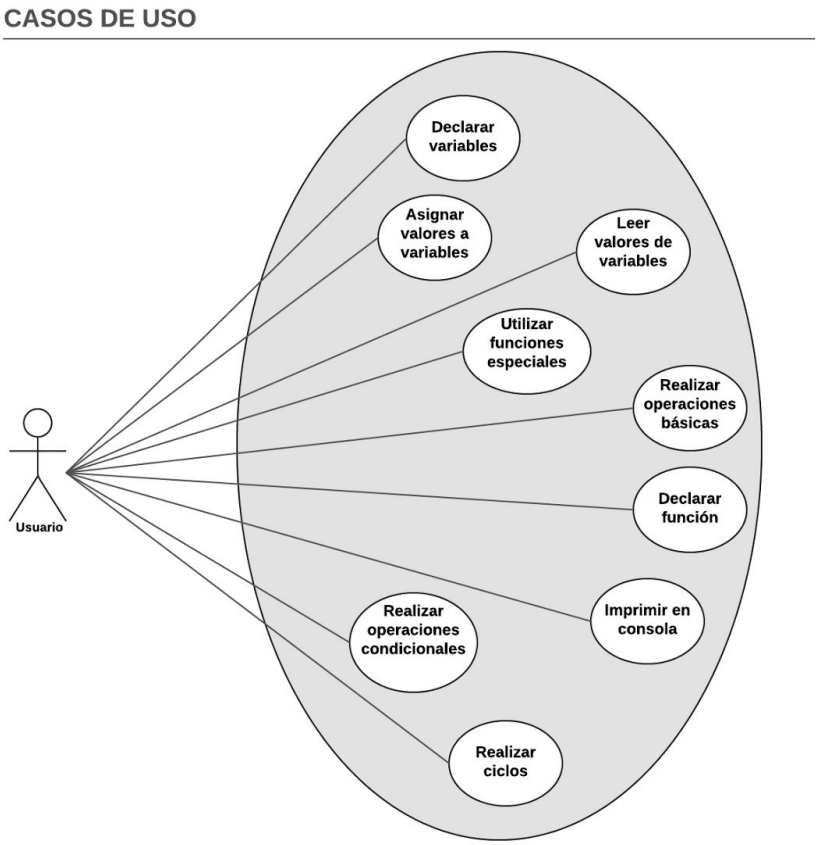
- Requerimientos funcionales

Número	Requerimiento
RF1	Se deberá poder declarar variables
RF2	Se deberá poder asignar valores a variables y arreglos
RF3	Se deberá poder obtener valores de variables y arreglos
RF4	Se deberán poder hacer operaciones básicas entre constantes, variables y arreglos
RF5	Se deberá poder declarar funciones
RF6	Se deberá poder soportar recursividad
RF7	Se deberá poder imprimir textualmente en consola
RF8	Se deberá poder realizar operaciones condicionales
RF9	Se deberá poder realizar ciclos
RF10	Se deberá poder utilizar las funciones especiales de música

- Requerimientos no funcionales

Número	Requerimiento
RNF1	La aplicación web debe poder ejecutarse en los navegadores más actuales
RNF2	Se carga correctamente las imágenes y la música
RNF3	El lenguaje Simphony debe compilar sin fallas

- Casos de Uso



1.1.3. Descripción de los principales Casos de Uso

Caso de Uso	Descripción
Factorial Iterativo	Programa que calcula el factorial de un número de manera iterativa
Factorial Recursivo	Programa que calcula el factorial de un número de manera recursiva
Fibonacci Iterativo	Programa que calcula el número “n” de la secuencia Fibonacci de manera iterativa
Fibonacci Recursivo	Programa que calcula el número “n” de la secuencia Fibonacci de manera recursiva

Bubble Sort	Programa que ordena un arreglo con el algoritmo de Bubble Sort
Find en arreglo	Programa que encuentra la posición de un valor en un arreglo
Ciclo musical	Programa que utiliza las funciones especiales de las notas musicales para crear una melodía

1.1.4. Descripción del proceso del desarrollo del proyecto

- Proceso general

Para desarrollar el proyecto utilizamos distintas herramientas de colaboración, como Google Docs para los documentos realizados, Github como controlador de versiones, así mismo utilizamos distintas redes sociales para mantener la comunicación entre nosotros. Durante la duración del proyecto tratamos siempre de trabajar conforme a los avances proporcionados por la profesora. Nos juntábamos en equipo por lo menos una vez por entrega de avance para discutir sobre lo que teníamos que hacer y cómo lo íbamos a desarrollar para posteriormente desarrollar lo que habíamos acordado.

- Bitácoras

BITÁCORA #1

NOMBRE DEL PROYECTO	Project Simphony
FECHA	Febrero / Marzo 27-3
AVANCE	#1
CONTENIDO ESPERADO	Análisis del léxico y sintaxis

CONTENIDO REAL
● Desarrollo del análisis léxico completo con pruebas exhaustivas (terminado) ● Desarrollo del análisis sintáctico parcial sin pruebas ○ Todos los diagramas han sido traducidos a una gramática formal, que ya está presente en los docstrings del código (tal como lo requiere PLY para su funcionamiento). ○ La gramática continúa teniendo ambigüedades al presentar conflictos shift-reduce y reduce-reduce. Se desconoce la causa de estas por el momento.

BITÁCORA #2

NOMBRE DEL PROYECTO	Project Simphony
FECHA	Marzo 6-10
AVANCE	#2
CONTENIDO ESPERADO	Semántica Básica de Variables: Directorio de Procedimientos y Tabla de Variables

CONTENIDO REAL
● Gramática y análisis sintáctico (terminado) ○ Después de una revisión con la profesora, se ha encontrado la causa de las ambigüedades y se han solucionado ○ Se añaden pruebas exhaustivas de gramática ○ La gramática ahora está completa y, por tanto, el análisis sintáctico también, concluyendo con lo esperado la semana pasada. ● Directorio de procedimientos ○ Ya se da soporte a ámbitos (scopes) globales y locales ○ Se proporcionan algunas pruebas ● Tabla de variables

○ Aún pendiente. No se añaden variables a los scopes todavía

BITÁCORA #3

NOMBRE DEL PROYECTO	Project Simphony
FECHA	Marzo 13-17
AVANCE	#3
CONTENIDO ESPERADO	Semántica Básica de Expresiones: Tabla de consideraciones semánticas (Cubo semántico) Generación de Código de Expresiones Aritméticas y estatutos secuenciales: Asignación, Lectura, etc.

CONTENIDO REAL
● Directorio de procedimientos (terminado) ○ Se añaden pruebas exhaustivas ● Tabla de variables (terminado) ○ Se registran correctamente las variables y sus propiedades ○ Se detectan variables no declaradas ○ Se añaden pruebas exhaustivas ● Cubo semántico (terminado) ○ Se ha diseñado e implementado ● Errores de tipo (terminado) ○ Ya se detectan los errores al utilizar variables ● Cuádruplos ○ Se está en proceso de producirlos, pero aún no funcionan ● Direcciones virtuales ○ Aún no se asignan. Se está diseñando la estructura de direcciones.

BITÁCORA #4

NOMBRE DEL PROYECTO	Project Simphony
FECHA	Marzo 20-24
AVANCE	#4
CONTENIDO ESPERADO	Generación de Código de Estatutos Condicionales: Decisiones/Ciclos

CONTENIDO REAL
● Cuádruplos de expresiones aritméticas (terminado) ○ Se generan los cuádruplos de expresiones aritméticas ○ Se proporcionan pruebas exhaustivas ● Direcciones virtuales (terminado) ○ Las direcciones ya se asignan ○ Se ha inspeccionado manualmente que se asignen correctamente ● Cuádruplos para estatutos condicionados ○ Aún no se generan. El equipo no avanzó en este requerimiento, pues decidió concentrar sus esfuerzos en terminarlo que estaba pendiente, que requirió un rediseño de varios módulos del compilador.

BITÁCORA #5

NOMBRE DEL PROYECTO	Project Simphony
FECHA	Marzo 27-31
AVANCE	#5
CONTENIDO ESPERADO	Generación de Código de Funciones

CONTENIDO REAL
● Cuádruplos para estatutos condicionados (terminado) ○ Se genera código y se hacen pruebas ● Cuádruplos de funciones ○ Se verifican los tipos de los argumentos ○ Falta el código de las demás secciones

BITÁCORA #6

NOMBRE DEL PROYECTO	Project Simphony
FECHA	Abril 3-7
AVANCE	#6
CONTENIDO ESPERADO	Mapa de Memoria de Ejecución de Máquina Virtual Máquina Virtual: Ejecución de Expresiones Aritméticas y Estatutos Secuenciales

CONTENIDO REAL
● Cuádruplos de funciones (terminado) ○ Se genera todo el código y se hacen pruebas exhaustivas ● Mapa de memoria (terminado) ○ Ya se tiene la memoria cargada con constantes y se puede accesar ● Ejecución de expresiones aritméticas y estatutos secuenciales ○ Pendiente. Ya se reconocen operaciones, pero falta ejecutarlas

BITÁCORA #7

NOMBRE DEL PROYECTO	Project Simphony
FECHA	Abril 17-21
AVANCE	#7
CONTENIDO ESPERADO	Generación de Código de Arreglos / Tipos Estructurados Máquina Virtual: Ejecución de Estatutos Condicionales

CONTENIDO REAL
● Ejecución de expresiones aritméticas y estatutos secuenciales (terminado) ○ Se ejecutan correctamente con los archivos de prueba ● Ejecución de estatutos condicionales (terminado) ○ Se ejecutan correctamente con los archivos de prueba ● Generación de código de arreglos / tipos estructurados (terminado) ○ Se generan los cuádruplos correctamente

BITÁCORA #8

NOMBRE DEL PROYECTO	Project Simphony
FECHA	Abril 24-28
AVANCE	#8
CONTENIDO ESPERADO	1era versión de la Documentación Generación de Código y Máquina Virtual para una parte de la aplicación en particular

CONTENIDO REAL
● 1era versión de la documentación (pendiente) ○ Documentación completa en un 65%. ● Ejecución de funciones (terminado) ○ Se ejecutan funciones (incluso recursivas) y se incluyen pruebas exhaustivas. Se manejan casos como funciones void que intentan usarse en una expresión. ● Ejecución de arreglos (terminado) ○ Se pueden utilizar arreglos, con el compilador validando todo lo necesario. Se incluyen pruebas exhaustivas. ● Portal web ○ Se termina la interfaz gráfica del portal y se avanza en cuanto a la funcionalidad del backend. Falta la integración formal con el compilador. ● EXTRA: incorporación de break (terminado) ○ Se le añade al proyecto el estatuto de break para salir de un ciclo, cosa que no estaba en la propuesta, pero se añade como un complemento.

- Reflexiones

Jorge Luis Márquez Sánchez

Este proyecto ha sido sin duda uno de los más retadores que hemos realizado en la carrera, ya que implica utilizar todos los conocimientos adquiridos durante los semestres anteriores. Para poder terminar este proyecto fue fundamental seguir los avances del proyecto propuestos por la profesora Elda, aunque a veces llegamos a retrasarnos un poco, nosotros por experiencia de otros compañeros sabíamos que teníamos que organizarnos para no tener que sufrir de más al final y poder terminar a tiempo.

La materia de Diseño de Compiladores ha sido una de las más difíciles de la carrera, puesto que aprendemos lo que hay detrás de cualquier lenguaje de programación. Sin embargo, creo que me llevo conocimientos que quizás pueda volver a utilizar en un futuro no muy lejano.

Firma:_____

Ali Ghahraei Figueroa
El proyecto de compiladores me pareció muy interesante y enriquecedor, ya que permite a los alumnos ver lo complejo que es tanto el diseño como la implementación de programas que utilizamos prácticamente todos los días y, sin los cuales, el desarrollo tecnológico actual no sería posible.
Considero que todo estudiante de una carrera relacionada con la programación debería hacer algo similar, pues con esto puede mejorar sus habilidades de administración de proyectos, ingeniería de software, análisis y diseño de algoritmos y muchas otras más.
La parte que más me gustó fue el desarrollo de la semántica, pues es la que hace el trabajo más pesado. Lo que menos me gustó fue la herramienta de PLY una vez que la conocí bien por sus limitaciones de diseño.

Firma:_____

1.2. DESCRIPCIÓN DEL LENGUAJE

1.2.1. Nombre del lenguaje

Symphony

1.2.2. Descripción genérica de las principales características del lenguaje

- Características del lenguaje

Declaración de variables: se pueden declarar variables locales y globales. Para declarar una variable solamente se escribe el tipo de dato y el nombre de la variable.
Ciclos: el lenguaje cuenta con el estatuto “while”, que actúa repetidamente mediante una expresión booleana definida dentro de sus paréntesis.
Condicionales: Se cuenta con el estatuto “if”, “elseif” y “else”, el cual se puede ejecutar por un solo camino si es que se cumple el “if”, si se tienen más casos se revisan con el “elseif” o en caso contrario se va por el “else”.
Escritura: El lenguaje cuenta con el estatuto “print” y “println”, el cual recibe cualquier variable o constante y la imprime en consola.

Funciones: Las funciones se deben declarar antes del bloque principal. Se llaman escribiendo los parámetros dentro de los paréntesis. Se soportan funciones con retorno de valor y sin retorno (void).
Arreglos: Simphony soporta la declaración de arreglos de una dimensión.
Expresiones: Se pueden realizar expresiones aritméticas, operaciones lógicas y de comparación.
Tipos de datos: Los tipos de dato que se soportan son entero, caracter, cadena de caracteres, booleano y decimal.

- Características del lenguaje musical

little_star: Toca la melodía de twinkle, twinkle little star.
C: Toca la nota de “Do” natural.
D: Toca la nota de “Re” natural.
E: Toca la nota de “Mi” natural.
F: Toca la nota de “Fa” natural.
G: Toca la nota de “Sol” natural.
A: Toca la nota de “La” natural.
B: Toca la nota de “Si” natural.

- Características del lenguaje matemático

sqrt: Permite obtener la raíz cuadrada de un valor.
log: Permite obtener el logaritmo natural de un valor.
random: Permite obtener un número aleatorio.
floor: Permite redondear un número hacia abajo.
ceil: Permite redondear un número hacia arriba.

- Características del lenguaje para cadenas de caracteres

length: Permite conocer el tamaño de una cadena de caracteres.
copy: Permite copiar una cadena de caracteres.
get: Permite obtener el caracter de la posición seleccionada de una cadena de caracteres.
to_str: Permite convertir un valor a una cadena de caracteres.

1.2.3. Descripción de los errores que pueden ocurrir, tanto en compilación como en ejecución

Mensaje	Razón
Error, la función no retorna ningún valor	Indica que se debió haber retornado algún valor de la función, pero no se devuelve ninguno.
Error, la función se declaró más de una vez	Indica que la función se está declarando más de una vez.
Error, la variable se encuentra ya declarada	Indica que la variable se está declarando más de una vez.
Error, la variable no fue declarada	Indica que la variable que se intenta acceder no fue declarada aún.
Error, condición de tipo incompatible	Indica que se está realizando una condicional con un valor de tipo incompatible.
Error, no puedes asignar un arreglo directamente	Indica que no se puede asignar un arreglo directamente.
Error de sintaxis	Indica que hay un error de sintaxis en el lenguaje Symphony.
Error, la función es void no tiene valor de retorno	Indica que se intento regresar un valor en una función de tipo void
Error, la función no fue declarada	Indica que se trató de llamar una función que aún no se encuentra declarada.
Error, la cantidad de parámetros no concuerda con la función	Indica que se enviaron parámetros de más o menos en una función.
Error, no se puede tener múltiples return	Indica que se trató de regresar múltiples valores en una función.
Error, se trató de acceder una	Indica que se trató de acceder una

variable pero no es un arreglo	variable, pero no era un arreglo.
Error, índice fuera de lo límites del arreglo	Indica que se intentó acceder un índice del arreglo fuera de sus límites.

1.3. DESCRIPCIÓN DEL COMPILADOR

1.3.1. Equipo de cómputo, lenguaje y utilerías especiales usadas en el desarrollo del proyecto

Los equipos de cómputo utilizados para realizar el compilador fueron una computadora HP corriendo Windows 7 y una Lenovo corriendo Arch Linux. Para el desarrollo del compilador utilizamos Python Lex Yacc (PLY) sobre Python 3.6.1.

1.3.2. Descripción del Análisis de Léxico

- Patrones de Construcción (expresados como Expresiones Regulares) de los elementos principales

Palabras reservadas				
program	int	dec	char	bool
str	void	return	if	elseif
else	while	and	or	not
equals	fun	mod	break	print
println	sqrt	log	random	little_star
A	B	C	D	E
F	G	concat	length	copy
get	to_str	floor	ceil	

Elementos de instancia única	
Token	Expresión Regular

literals	[',', ';', '(', ')', '{', '}', '[', ']', '=', '+', '-', '*', '/', '>', '<',]
t_GREATER_EQUAL_THAN	r'>='
t_LESS_EQUAL_THAN	r'<='
t_EXPONENTIATION	r'**'
t_INCREMENT	r'\+\+'
t_DECREMENT	r'--'

- Enumeración de los “tokens” del lenguaje y su código asociado

```
CUBE = [
    [
        [Types.INT] * 3 + [Types.DEC] + [Types.INT] * 2 + [Types.BOOL] * 5,
        [],
        [],
        [],
        [Types.DEC] * 6 + [Types.BOOL] * 5,
    ],
    [
        [],
        [Types.STR] + [None] * 5 + [Types.BOOL] * 5,
        [Types.STR],
        [],
        [],
    ],
    [
        [],
        [Types.STR],
        [Types.STR] + [None] * 5 + [Types.BOOL] * 5,
        [],
        [],
    ],
    [
        [],
        [],
        [],
        [None] * 6 + [Types.BOOL] * 8,
        [],
    ],
    [
```

```
        [Types.DEC] * 6 + [Types.BOOL] * 5,
        [],
        [],
        [],
        [Types.DEC] * 6 + [Types.BOOL] * 5,
    ],
]

UNARY_TABLE = [
    [Types.INT.value] * 4,
    [],
    [],
    [None] *4 + [Types.BOOL.value],
    [Types.DEC.value] *4,
]
```

1.3.3. Descripción del Análisis de Sintaxis

- Grámatica Formal

Gramática Formal
program : PROGRAM ID ';' variable_declaration function_declaration block empty : variable_declaration : type ids ';' variable_declaration | empty ids : id other_ids other_ids : ',' ids | empty id : ID | ID '[' expression ']' expression : level1 | level1 EXPONENTIATION level1 level1 : level2 | '+' level2 | '-' level2 level2 : level3

```
        | level3 OR level3
        | level3 AND level3

level3 : level4
        | level4 '<' level4
        | level4 '>' level4
        | level4 LESS_EQUAL_THAN level4
        | level4 GREATER_EQUAL_THAN level4
        | level4 EQUALS level4

level4 : level5
        | level5 '+' level5
        | level5 '-' level5

level5 : level6
        | NOT level6
        | level6 '*' level6
        | level6 '/' level6
        | level6 MOD level6

level6 : '(' expression ')'
        | const
        | increment
        | decrement

increment : INCREMENT id

decrement : DECREMENT id

function_declaration : function function_declaration
        | empty

'"function : FUN return_type ID '(' parameters ')' '{' variable_declaration statutes
'}' ';'

return_type : type
        | VOID

type : INT
        | DEC
        | CHAR
        | STR
        | BOOL

statutes : statute ';' statutes
        | empty
```

```
'''statute   : call
             | assignment
             | condition
             | cycle
             | special
             | return
             | increment
             | decrement

call : ID '(' expressions ')'

expressions : expression
             | expression ',' expressions

assignment : id '=' expression

condition : IF '(' expression ')' block elses

cycle : WHILE '(' expression ')' block

special : SPECIAL_ID '(' expressions ')'

return : RETURN expression
             | RETURN

elses : empty
             | ELSE block
             | ELSEIF '(' expression ')' block elses

parameters : type ID other_parameters
             | empty

other_parameters : ',' type ID other_parameters
             | empty

const : id
             | call
             | special
             | INT_VAL
             | DEC_VAL
             | CHAR_VAL
             | STR_VAL
             | BOOL_VAL

block : '{' statutes '}'
```

1.3.4. Descripción de Generación de Código Intermedio y Análisis Semántico

- Código de operación y direcciones virtuales asociadas a los elementos del código

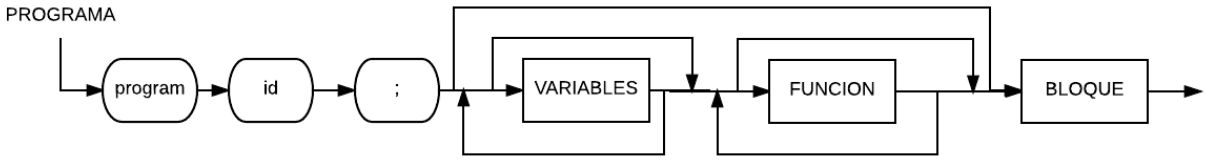
Código de operación	Descripción
add	Operación aritmética de suma
sub	Operación aritmética de resta
mult	Operación aritmética de multiplicación
truediv	Operación aritmética de división
pow	Operación aritmética de exponenciación
mod	Operación aritmética de modulo
eq	Operación de comparación igual
gt	Operación de comparación mayor que
lt	Operación de comparación menor que
ge	Operación de comparación mayor igual que
le	Operación de comparación menor igual que
and_	Operación lógica and
or_	Operación lógica or
partial(add, 1)	Operación de incremento
lambda value: sub(value, 1)	Operación de decremento
pos	Número positivo
neg	Número negativo
not_	Operación lógica not
lambda value: value	Operación de asignación
store_param	Guarda los parámetros

partial(print_, end='')	Imprime en la pantalla
print_	Imprime en la pantalla con salto de línea
sqrt_	Operación de raíz cuadrada
log_	Operación de logaritmo natural
get	Obtiene la entrada
little_star	Función de twinkle, twinkle, little star
A	Función de “La” natural
B	Función de “Si” natural
C	Función de “Do” natural
D	Función de “Re” natural
E	Función de “Mi” natural
F	Función de “Fa” natural
G	Función de “Sol” natural
length	Función del tamaño de cadena de caracteres
copy	Función que copia una cadena de caracteres
random_	Función que genera un número aleatorio decimal
to_str	Función que convierte un valor a una cadena de caracteres
input_	Obtiene la entrada
floor_	Función que redondea un valor hacia abajo
ceil_	Función que redondea un valor hacia arriba
goto	Indica un brinco de cuádruplo
gotof	Si es falso, indica un brinco de cuádruplo
array_access	Acceso al arreglo
verify_limits	Verifica los límites del arreglo
gosub	Envía a dormir la memoria actual

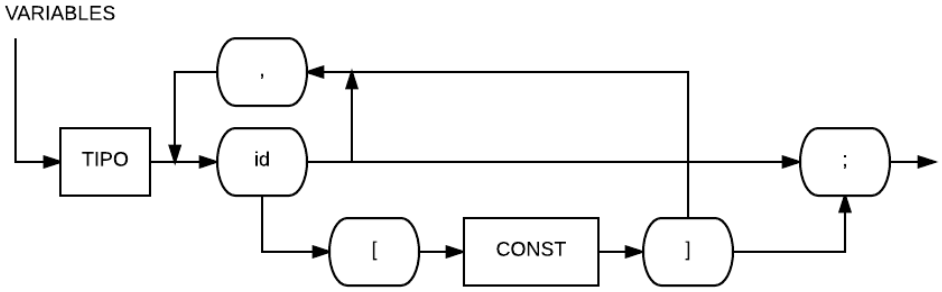
end_proc	Indica que se ha terminado una función

Sector de memoria	Memoria
global_	10_000
temporal	130_000
constant	200_000
local	250_000
end	350_000

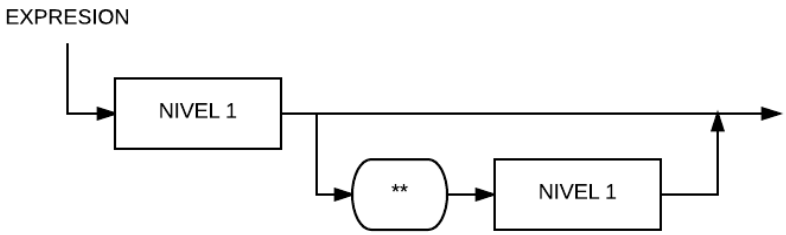
- Diagramas de Sintaxis con las acciones correspondientes



- Rellena cuádruplo de goto
- Genera cuádruplo de fin de programa

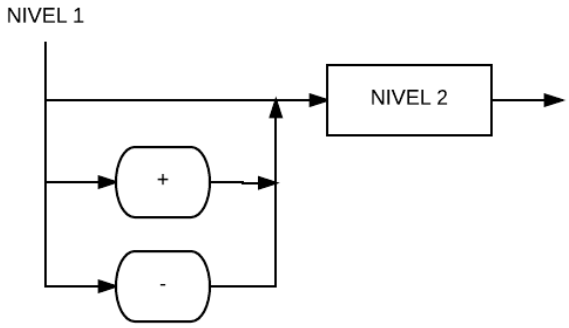


- Enciende bandera de arreglo
- Verifica que ese id no esté ya registrado en la tabla de variables

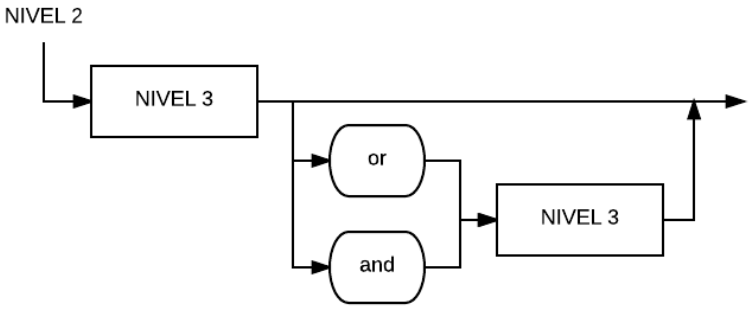


- Push signo a la pila de operadores

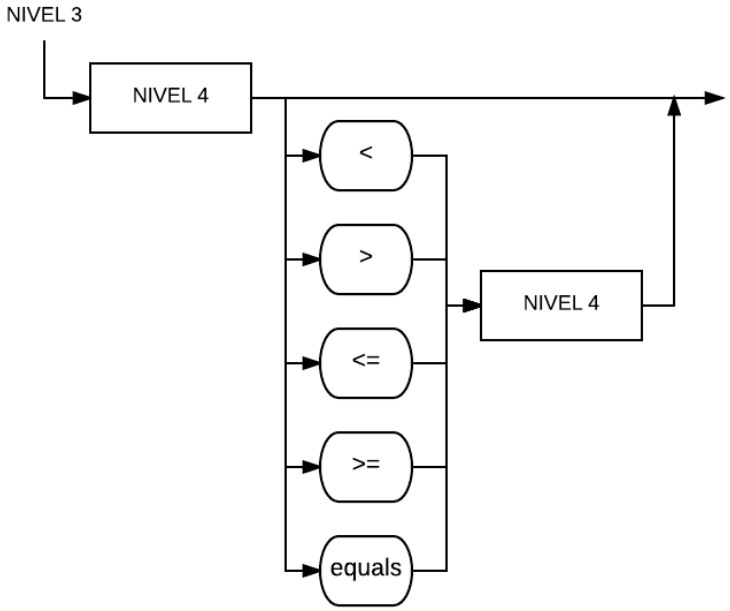
- Mientras la pila de operadores tenga ** el vector polaco es = pop de pila de operadores



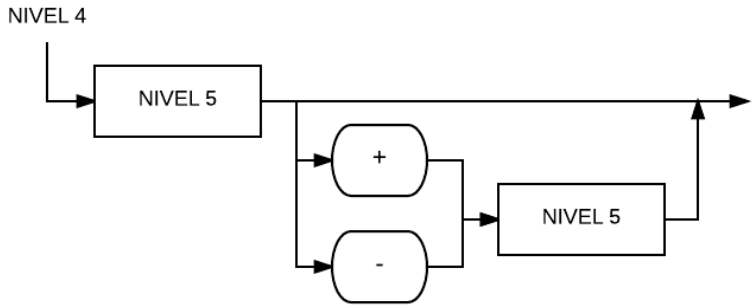
- Push signo a la pila de operadores
- Mientras la pila de operadores tenga + o - el vector polaco es = pop de pila de operadores

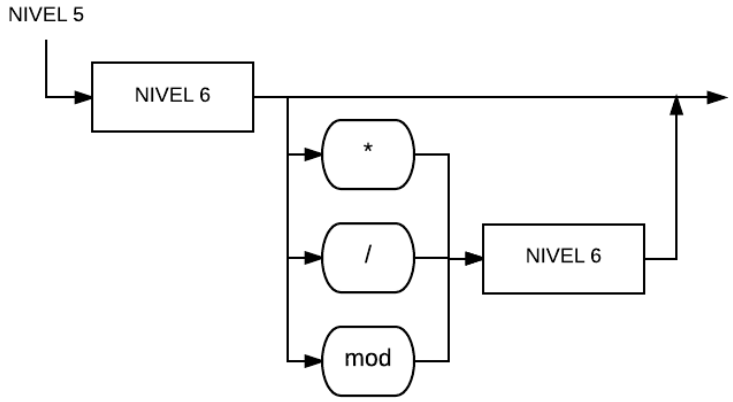


- Push signo a la pila de operadores
- Mientras la pila de operadores tenga or o and el vector polaco es = pop de pila de operadores

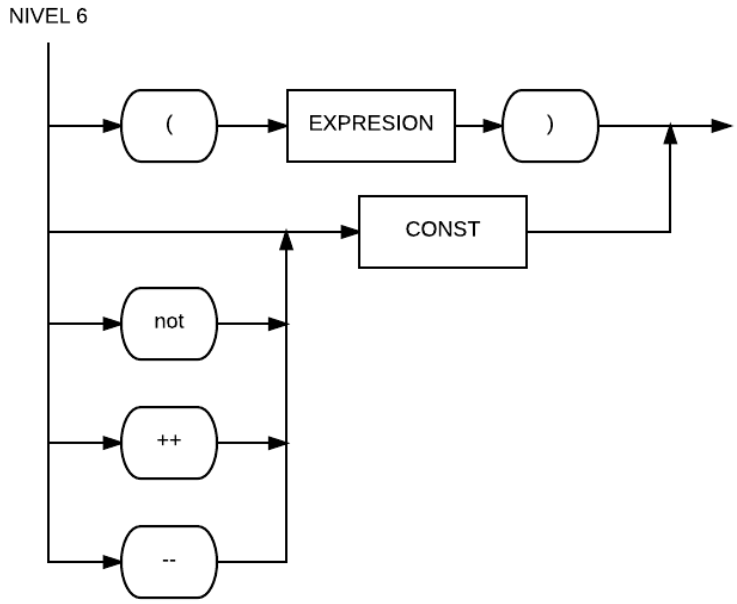


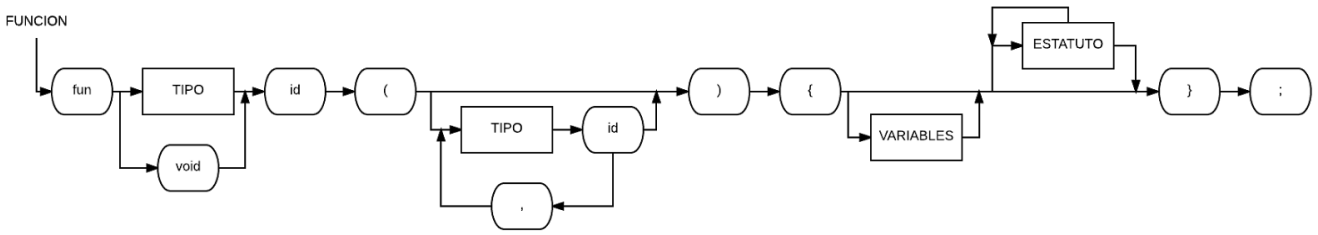
- Push signo a la pila de operadores
- Mientras la pila de operadores tenga `<, >, <=, >=, equals` el vector polaco es = pop de pila de operadores



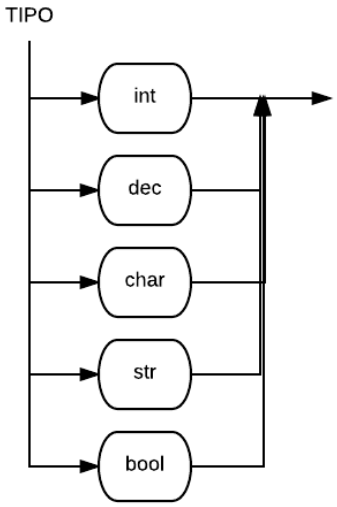


- Push signo a la pila de operadores
- Mientras la pila de operadores tenga *, / o mod el vector polaco es = pop de pila de operadores

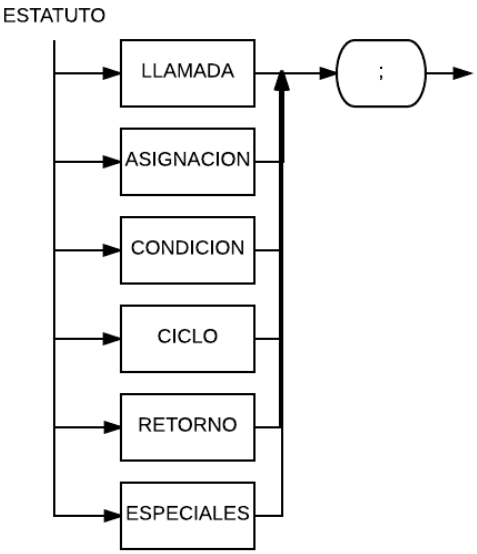




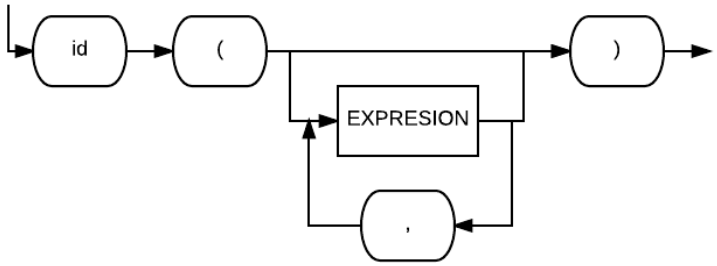
- Guarda el tipo de la función
- Obtiene y guarda el nombre de la función
- Genera cuádruplo de fin de función



- Guarda el último tipo

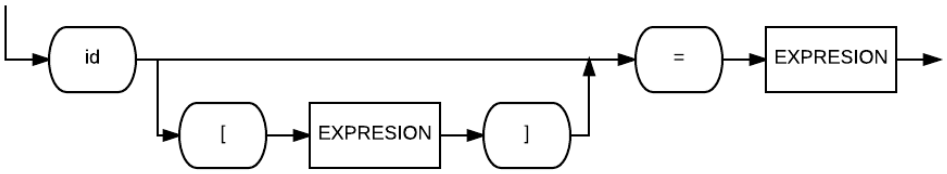


LLAMADA



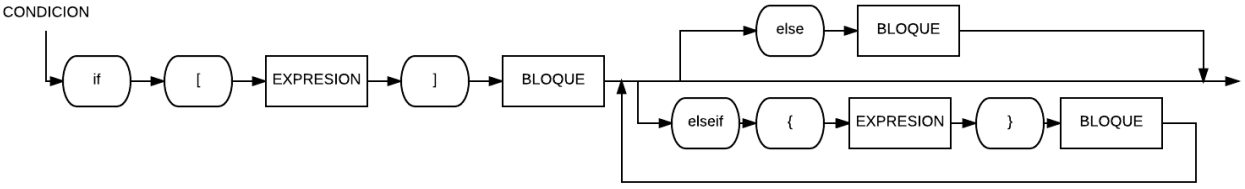
- Revisa que la función exista en el directorio de procedimientos
- Genera cuádruplo para la función

ASIGNACION



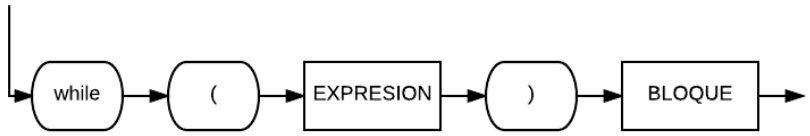
- Verifica que la variable exista
- Enciende la bandera de arreglo
- Revisa que el tipo de la expresión sea numérico
- Revisa que el tipo de la expresión o función sea asignable

CONDICION



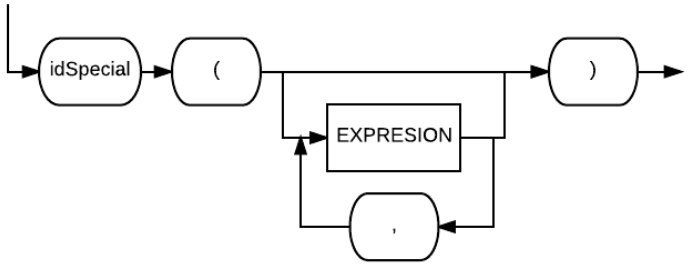
- Revisa que el tipo de la expresión sea booleana y genera cuádruplos de if
- Genera cuádruplos de elseif (si se requiere)
- Genera cuádruplos else
- Genera fin de condicional y rellena cuádruplos

CICLO



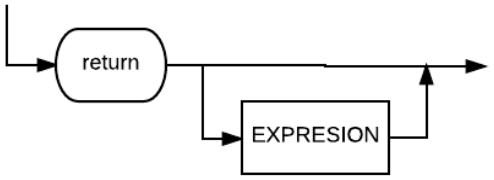
- Genera cuádruplos y mete a pila de saltos
- Revisa que el resultado de la expresión sea booleano
- Genera goto al inicio de la expresión

ESPECIALES



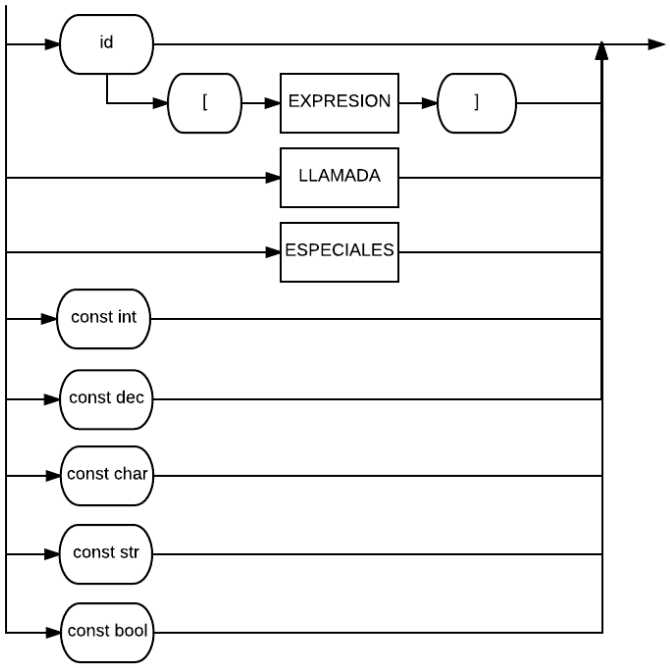
- Revisa que el número de expresiones sea el correcto

RETORNO



- Revisa que el tipo de la expresión sea el mismo que el tipo de la función

CONST



- Enciende bandera de arreglo
- Revisa que el tipo de la expresión sea entero
- Metemos el tipo a la pila de tipos

BLOQUE

- Tabla de consideraciones semánticas

Op. 1	Op. 2	+	-	*	/	**	mod	equals	>	<	>=	<=	and	or	not
int	int	int	int	int	dec	int	int	bool	bool	bool	bool	bool	ERR	ERR	ERR
int	char	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
int	str	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
int	bool	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
int	dec	dec	dec	dec	dec	dec	dec	bool	bool	bool	bool	bool	ERR	ERR	ERR
char	int	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
char	char	str	ERR	ERR	ERR	ERR	ERR	bool	bool	bool	bool	bool	ERR	ERR	ERR
char	str	str	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
char	bool	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
char	dec	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
str	int	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
str	char	str	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
str	str	str	ERR	ERR	ERR	ERR	ERR	bool	bool	bool	bool	bool	ERR	ERR	ERR
str	bool	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
str	dec	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
bool	int	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
bool	char	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
bool	str	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
bool	bool	ERR	ERR	ERR	ERR	ERR	ERR	bool	bool	bool	bool	bool	bool	bool	bool
bool	dec	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
dec	int	dec	dec	dec	dec	dec	dec	bool	bool	bool	bool	bool	ERR	ERR	ERR
dec	char	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
dec	str	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
dec	bool	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
dec	dec	dec	dec	dec	dec	dec	dec	bool	bool	bool	bool	bool	ERR	ERR	ERR

1.3.5. Descripción detallada del proceso de Administración de Memoria usado en la compilación

- Especificación textual de cada estructura de datos usada

En compilación, el parser utiliza una estructura de datos proveída por la máquina virtual, llamada tupla de direcciones (Address Tuple). La estructura es utilizada, en términos generales, para llevar la cuenta de las direcciones que han sido asignadas a cada grupo de memoria (global, local, constante y temporal). La tupla es una tupla nombrada, por lo que puede accederse a los sectores por nombre. Cada uno de estos tiene además dónde empieza cada tipo para manejar la asignación con mayor granularidad. Adicionalmente, la tupla puede crearse con las direcciones donde terminan si se pasa un parámetro específico a la función que la genera.
También se utiliza un objeto directorio que guarda referencias a objetos FunctionScope para almacenar los diferentes contextos. Los FunctionScope tienen tipo de retorno, un diccionario de variables, tipos de parámetro junto con sus direcciones (que son una cola para invertir el orden), cuádruplo en donde inician y dirección de retorno.

El objeto más importante en compilación es el generador de cuádruplos, pues este conserva cuádruplos para su modificación por los saltos pendientes, mantiene las direcciones de constantes, lista las funciones llamadas y registra las invocaciones recursivas, entre otras. El objeto contiene un gran número de funciones y es el principal intermediario entre la sintaxis y la semántica.

1.4. DESCRIPCIÓN DE LA MÁQUINA VIRTUAL

1.4.1. Equipo de cómputo, lenguaje y utilerías especiales usadas (en caso de estar en diferente que el compilador)

Los equipos de cómputo utilizados para realizar el compilador fueron una computadora HP corriendo Windows 7 y una Lenovo

corriendo Arch Linux. Para el desarrollo del compilador utilizamos Python Lex Yacc (PLY) sobre Python 3.6.1.

1.4.2. Descripción detallada del proceso de Administración de Memoria en ejecución (Arquitectura). Incluir:

- Especificación gráfica de CADA estructura de datos usada (Memoria Local, global, etc…)

  - Memory
  - Activation Records
  - Stored Program Counters
  - Parameters
  - Output
  - Addresses

- Asociación hecha entre las direcciones virtuales (compilación) y las reales (ejecución)

  Las direcciones se mantienen sincronizadas gracias a la tupla que genera la máquina virtual y le retorna al parser. Como ya se mencionó anteriormente, en compilación se asignan direcciones manteniendo la cuenta de en qué dirección va cierto tipo de cierto sector. Las direcciones se recuperan tal y como se indicó en los cuádruplos con funciones de almacenaje y recuperación, empleando los rangos generados por la tupla compartida.

1.5. PRUEBAS DEL FUNCIONAMIENTO DEL LENGUAJE

1.5.1. Incluir pruebas que “comprueben” el funcionamiento del proyecto:

- Codificación de la prueba (en su lenguaje)

Codificación del programa factorial iterativo

```
program factorial_iter;
    int number, result, i;
    number = 20;

    i = 1;
    result = 1;
    while(i <= number) {
        result = result * i;
        ++i;
    }

    println(result);

```

Codificación del programa factorial recursivo

```
program factorial;
    fun int factorial(int number){
        int result;

        if((number equals 0) or (number equals 1)){
            result = 1;
        } else {
            result = number * factorial(number - 1);
        }

        return result;
    }
    print(factorial(10) + factorial(4));
```

Codificación del programa fibonacci iterativo

```
program fibonacci_iter;
    int result, term;
    int one_before, two_before, i, sum, t;

    term = 8;

    if(term <= 0 or term equals 1) {
        result = 0;
    } else {
        one_before = 1;
        two_before = 0;

        i = 2;
        while(not (i equals term)) {
            t = two_before;
            two_before = one_before;
```

```
        one_before = one_before + t;
        ++i;
    }

    result = one_before;
}
println(result);
```

Codificación del programa fibonacci recursivo

```
program fibonacci;
    fun int fibonacci(int term) {
        int result;

        if(term <= 0) {
            result = 0;
        } elseif(term equals 1) {
            result = 0;
        } elseif(term equals 2) {
            result = 1;
        } else {
            result = fibonacci(term - 1) + fibonacci(term - 2);
        }

        return result;
    }
    print(fibonacci(9));
```

Código del programa ordenamiento de burbuja

```
program bubble_sort;
    int i, elements[3];
    fun void sort(int size) {
        int unsorted, comparisons_done, t;
        bool swapped;

        unsorted = size;
        while(unsorted > 1) {
            comparisons_done = 0;
            swapped = false;

            while(comparisons_done < unsorted - 1) {
```

```
            if(elements[comparisons_done] >
elements[comparisons_done + 1]) {
                t = elements[comparisons_done];
                elements[comparisons_done] =
elements[comparisons_done + 1];
                elements[comparisons_done + 1] = t;
                swapped = true;
            }
            ++comparisons_done;
        }

        if(not swapped) {
            break;
        }

        --unsorted;
    }
}

elements[0] = 7;
elements[1] = 2;
elements[2] = 4;

sort(3);

i = 0;
while(i < 3) {
    print(elements[i]);
    print(" ");
    ++i;
}
println("");
```

Código del programa encontrar un elemento en un vector

```
rogram insertion_sort_and_search;
    int elements[5], i;
    fun void sort(int size) {
        int i, j, t;
        i = 1;

        while(i <= size - 1) {
            j = i;
```

```
        while (j > 0) {
            if(elements[j] < elements[j - 1]) {
                break;
            }

            t = elements[j];
            elements[j] = elements[j - 1];
            elements[j - 1] = t;

            --j;
        }

        ++i;
    }
}


fun int binary_search(int start, int end, int wanted) {
    int half, result;
    half = floor((start + end) / 2);

    if(end < start) {
        result = -1;
    } elseif(elements[half] equals wanted) {
        result = half;
    } elseif(wanted > elements[half]) {
        result = binary_search(start, half - 1, wanted);
    } else {
        result = binary_search(half + 1, end, wanted);
    }

    return result;
}


elements[0] = 6;
elements[1] = 0;
elements[2] = 157;
elements[3] = 4;
elements[4] = -10;

sort(5);
```

```
   i = 0;
   while(i < 5) {
      print(elements[i]);
      print(' ');
      ++i;
   }
   println("");

   println("Found 4 at index: " + to_str(binary_search(0, 4, 4)));
   println("Found 157 at index: " + to_str(binary_search(0, 4,
157)));
   println("Found 0 at index: " + to_str(binary_search(0, 4, 0)));
   println("Found 20 at index: " + to_str(binary_search(0, 4, 20)));
```

- Resultados arrojados por la generación de código intermedio y por la ejecución

Resultado de la generación de código intermedio del programa factorial iterativo

```
GOTO 1
= 200000 10002
= 200001 10000
= 200001 10001
<= 10000 10002 172000
GOTOF 172000 10
* 10001 10000 130000
= 130000 10001
++ 10000 10000
GOTO 4
PARAM 10001 1
println
```

Resultado de la ejecución del programa factorial iterativo

```
2432902008176640000
```

Resultado de la generación de código intermedio del programa factorial recursivo

```
GOTO 14
equals 250000 200000 172000
equals 250000 200001 172001
or 172000 172001 172002
GOTOF 172002 7
= 200001 250001
GOTO 13
- 250000 200001 130000
PARAM 130000 1
GOSUB factorial
= 250001 250002
* 250000 250002 130001
= 130001 250001
ENDPROC factorial
PARAM 200002 1
GOSUB factorial
= 250001 10000
PARAM 200003 1
GOSUB factorial
= 250001 10001
+ 10000 10001 130002
PARAM 130002 1
print
```

Resultado de la ejecución del programa factorial recursivo

```
3628824
```

Resultado de la generación de código intermedio del programa fibonacci iterativo

```
GOTO 1
```

```
= 200000 10000
<= 10000 200001 172000
equals 10000 200002 172001
or 172000 172001 172002
GOTOF 172002 8
= 200001 10001
GOTO 21
= 200002 10006
= 200001 10005
= 200003 10004
equals 10004 10000 172003
not 172003 172004
GOTOF 172004 20
= 10005 10002
= 10006 10005
+ 10006 10002 130000
= 130000 10006
++ 10004 10004
GOTO 11
= 10006 10001
PARAM 10001 1
println
```

Resultado de la ejecución del programa fibonacci iterativo

```
13
```

Resultado de la generación de código intermedio del programa fibonacci recursivo

```
GOTO 24
<= 250000 200000 172000
GOTOF 172000 5
= 200000 250001
GOTO 23
equals 250000 200001 172001
GOTOF 172001 9
= 200000 250001
GOTO 23
equals 250000 200002 172002
GOTOF 172002 13
= 200001 250001
```

```
GOTO 23
- 250000 200001 130000
PARAM 130000 1
GOSUB fibonacci
= 250001 250002
- 250000 200002 130001
PARAM 130001 1
GOSUB fibonacci
= 250001 250003
+ 250002 250003 130002
= 130002 250001
ENDPROC fibonacci
PARAM 200003 1
GOSUB fibonacci
= 250001 10000
PARAM 10000 1
print
```

Resultado de la ejecución del programa fibonacci recursivo

```
21
```

Resultado de la generación de código intermedio del programa ordenamiento de burbuja

```
GOTO 38
= 250000 250003
> 250003 200001 172000
GOTOF 172000 37
= 200002 250002
= 230000 310000
- 250003 200001 130000
< 250002 130000 172001
GOTOF 172001 32
VER 250002 0 3
ACCESS 10000 250002 130001
+ 250002 200001 130002
VER 130002 0 3
ACCESS 10000 130002 130003
> &130001 &130003 172002
GOTOF 172002 30
```

```
VER 250002 0 3
ACCESS 10000 250002 130004
= &130004 250001
+ 250002 200001 130005
VER 130005 0 3
ACCESS 10000 130005 130006
VER 250002 0 3
ACCESS 10000 250002 130007
= &130006 &130007
+ 250002 200001 130008
VER 130008 0 3
ACCESS 10000 130008 130009
= 250001 &130009
= 230001 310000
++ 250002 250002
GOTO 6
not 310000 172003
GOTOF 172003 35
GOTO 37
-- 250003 250003
GOTO 2
ENDPROC sort
VER 200002 0 3
ACCESS 10000 200002 130010
= 200003 &130010
VER 200001 0 3
ACCESS 10000 200001 130011
= 200004 &130011
VER 200004 0 3
ACCESS 10000 200004 130012
= 200005 &130012
PARAM 200000 1
GOSUB sort
= 200002 10003
< 10003 200000 172004
GOTOF 172004 60
VER 10003 0 3
ACCESS 10000 10003 130013
PARAM &130013 1
print
PARAM 220000 1
print
++ 10003 10003
GOTO 50
```

```
PARAM 220001 1
println
```

Resultado de la ejecución del programa ordenamiento de burbuja

```
2 4 7
```

Resultado de la generación de código intermedio del programa encontrar en un elemento en un vector

```
GOTO 69
= 200001 250003
- 250000 200001 130000
<= 250003 130000 172000
GOTOF 172000 33
= 250003 250002
> 250002 200002 172001
GOTOF 172001 31
VER 250002 0 5
ACCESS 10001 250002 130001
- 250002 200001 130002
VER 130002 0 5
ACCESS 10001 130002 130003
< &130001 &130003 172002
GOTOF 172002 16
GOTO 31
VER 250002 0 5
ACCESS 10001 250002 130004
= &130004 250001
- 250002 200001 130005
VER 130005 0 5
ACCESS 10001 130005 130006
VER 250002 0 5
ACCESS 10001 250002 130007
= &130006 &130007
- 250002 200001 130008
VER 130008 0 5
ACCESS 10001 130008 130009
= 250001 &130009
-- 250002 250002
GOTO 6
```

```
++ 250003 250003
GOTO 2
ENDPROC sort
+ 250006 250005 130010
/ 130010 200003 186000
PARAM 186000 1
floor 130011
= 130011 250008
< 250005 250006 172003
GOTOF 172003 43
= 200004 250007
GOTO 68
VER 250008 0 5
ACCESS 10001 250008 130012
equals &130012 250004 172004
GOTOF 172004 49
= 250008 250007
GOTO 68
VER 250008 0 5
ACCESS 10001 250008 130013
> 250004 &130013 172005
GOTOF 172005 61
- 250008 200001 130014
PARAM 250006 1
PARAM 130014 2
PARAM 250004 3
GOSUB binary_search
= 250007 250009
= 250009 250007
GOTO 68
+ 250008 200001 130015
PARAM 130015 1
PARAM 250005 2
PARAM 250004 3
GOSUB binary_search
= 250007 250010
= 250010 250007
ENDPROC binary_search
VER 200002 0 5
ACCESS 10001 200002 130016
= 200005 &130016
VER 200001 0 5
ACCESS 10001 200001 130017
= 200002 &130017
```

```
VER 200003 0 5
ACCESS 10001 200003 130018
= 200006 &130018
VER 200007 0 5
ACCESS 10001 200007 130019
= 200008 &130019
VER 200008 0 5
ACCESS 10001 200008 130020
= 200009 &130020
PARAM 200000 1
GOSUB sort
= 200002 10000
< 10000 200000 172006
GOTOF 172006 97
VER 10000 0 5
ACCESS 10001 10000 130021
PARAM &130021 1
print
PARAM 210000 1
print
++ 10000 10000
GOTO 87
PARAM 220000 1
println
PARAM 200002 1
PARAM 200008 2
PARAM 200008 3
GOSUB binary_search
= 250007 10006
PARAM 10006 1
to_str 158000
+ 220001 158000 158001
PARAM 158001 1
println
PARAM 200002 1
PARAM 200008 2
PARAM 200006 3
GOSUB binary_search
= 250007 10007
PARAM 10007 1
to_str 158002
+ 220002 158002 158003
PARAM 158003 1
println
```

```
PARAM 200002 1
PARAM 200008 2
PARAM 200002 3
GOSUB binary_search
= 250007 10008
PARAM 10008 1
to_str 158004
+ 220003 158004 158005
PARAM 158005 1
println
PARAM 200002 1
PARAM 200008 2
PARAM 200010 3
GOSUB binary_search
= 250007 10009
PARAM 10009 1
to_str 158006
+ 220004 158006 158007
PARAM 158007 1
println
```

Resultado de la ejecución del programa encontrar un elemento en un vector

```
157 6 4 0 -10
Found 4 at index: 2
Found 157 at index: 0
Found 0 at index: 3
Found 20 at index: -1
```

1.6. LISTADOS PERFECTAMENTE DOCUMENTADOS DEL PROYECTO

1.6.1. Incluir comentarios de **Documentación**, es decir: para cada módulo, una pequeña explicación de qué hace, qué parámetros recibe, qué genera como salida y cuáles son los módulos más importantes que hacen uso de él.

1.6.2. Dentro de los módulos principales se esperan comentarios de **Implementación**, es decir: pequeña descripción de cuál es la función de algún estatuto que sea importante de ese módulo

Declaración de variables en masa, de funciones y de variables individuales

```
    def declare_variables(self, parameters, variables, line_number,
                          is_global=False):
        """Invoked to declare groups of variables in functions (even
global)

        This function declares variables in 'bundles' collected in a
function
        definition. It declares parameters and variables. If one wishes to
        define the global scope, a keyword can be passed
        """
        # Store where the function starts
        self.functions[self.current_scope].first_quadruple = len(
            quadruple_generator.quadruples)

        for parameter in parameters:
            # parameter[0] has its type, which is stored in the signature

self.functions[self.current_scope].parameter_types.appendleft(
                parameter[0])

            # Declare the actual variable and also store it in parameters
            self._declare_variable(parameter, is_global, line_number)

self.functions[self.current_scope].parameter_addresses.appendleft(

self.functions[self.current_scope].variables[parameter[1]][1])

        # Declare each normal variable
        for variable in variables:
            self._declare_variable(variable, is_global, line_number)


    def define_function(self, return_type, function, line_number):
        """ Define an individual function """
        # Reject multiple declarations
        if function in self.functions:
            raise RedeclarationError(f'Error on line {line_number}: you
are'
                                     f' defining your {function} function more'
                                     f' than once')

        # Create a new function with a starting quad in the current quad
        starting_quad = len(quadruple_generator.quadruples)
        self.current_scope = function
        self.functions[function] = FunctionScope(return_type, function,
                                                 starting_quad)


    def _declare_variable(self, variable, is_global, line_number):
        """ Declare individual variables """
        current_function_vars =
self.functions[self.current_scope].variables

        try:
            # Unpack variable and array size
            variable_type, (variable_name, array_size) = variable
            array_size_type, array_size_value = array_size
        except ValueError:
            # An exception means that the variable is not an array
            variable_type, variable_name = variable

            if variable_name in current_function_vars:
                raise RedeclarationError(f'Error on line {line_number}: you
are '
                                         f'declaring your {variable_name} '
                                         f' variable more than once')

            # Create a non-dimensional variable
            current_function_vars[variable_name] = (
                variable_type,

quadruple_generator.generate_variable_address(variable_type,
                                                      is_global),
                variable_name,
            )
        else:
            if variable_name in current_function_vars:
                raise RedeclarationError(f'Error on line {line_number}: you
are '
                                         f'declaring your {variable_name} '
                                         f'variable more than once')


            # Validate array size type
            if array_size_type != Types.INT:
                raise TypeError(f'Error on line {line_number}: you are trying
to'
                                f' declare an array size using a(n) '
```

```
                    f'{array_size_type.name}, but you should use '
                    f'a(n) {Types.INT.name} instead')

        # Create a dimensional variable
        current_function_vars[variable_name] = (
            NonUserTypes.ARRAY,
            quadruple_generator.generate_variable_address(
                variable_type,
                is_global,
                array_size_value
            ),
            variable_name,
            variable_type,
            array_size_value
        )
```

Generación de acceso a variables dimensionadas

```
    def generate_access(self, array_name, line_number):
        """ Generate an array access """
        offset_type, offset_value = self.pop_operand(line_number)

        if offset_type != Types.INT:
            raise TypeError(f'Error on line {line_number}: you are trying to'
                            f' access an array using a(n) '
                            f'{offset_type.name}, but you should use '
                            f'a(n) {Types.INT.name} instead')

        variable = directory.get_variable(array_name, line_number)

        type_ = variable[0]

        if type_ != NonUserTypes.ARRAY:
            raise TypeError(f"Error on line {line_number}: you tried to access "
                            f"your {array_name} variable, but it's not an array")

        array_size_value = variable[4]
        self.generate_quad('VER', offset_value, 0, array_size_value)

        base_address = variable[1]
        real_type = variable[3]
        result_address = self.generate_temporal_address(real_type)

        self.generate_quad('ACCESS', base_address, offset_value, result_address)
        return real_type, "&" + str(result_address)
```

Generación de scopes de memoria (gosub) y eliminación (endproc)

```
def end_proc(function_name):
    """ Finish a function definition, Restoring a previous activation record """
    return_address = directory.functions[function_name].return_address
    if return_address != None:
        return_type = directory.functions[function_name].return_type
        try:
            # Copy the return value of a context into the previous one
            return_value = memory['local'][return_type][return_address]
            activation_records[-1][return_type][return_address] = return_value
        except KeyError:
            pass

    # Restore the previous context
    memory['local'] = activation_records.pop()
    return stored_program_counters.pop() + 1


def gosub(function_name):
    """ Save the current activation record and trigger a context change """
    activation_records.append(deepcopy(memory['local']))

    global directory
    function = directory.functions[function_name]
    # Load each argument into the function's new context
    for type_, address, argument in zip(function.parameter_types,
                                        function.parameter_addresses,
                                        parameters):
        memory['local'][type_][address] = value(argument)

    parameters.clear()
    raise ChangeContext(function.starting_quad)
```

Generación de direcciones virtuales para parser y máquina virtual

```
def generate_memory_addresses(end_addresses=False):
    """Generate a tuple of memory addresses

    This function is used by the VM and the parser. It returns the starting and
    ending address of every sector and type to allow a fast lookup of what the
    type of an address is and to count how many variables of the same type have
    been assigned per sector
    """
    ADDRESS_TUPLE = namedtuple('ADDRESSES', [address[0] for address
                                    in MEMORY_SECTORS[:-1]])

    # Begin with the global starting address
    current_address = MEMORY_SECTORS[0][1]
    addresses = []
    for sector in MEMORY_SECTORS[1:]:
        # Get the address of the next sector to calculate sector sizes
        next_address = sector[1]
        sector_size = next_address - current_address

        # Determine where the address of each type in the sector starts
        type_start_addresses = [starting_address for starting_address in
                            range(current_address, next_address,
                                int(sector_size / len(Types)))]

        # Generate an additional field indicating the end of each type
        # address if requested by the caller
        if end_addresses:
            type_end_addresses = type_start_addresses[1:] + [next_address]

            type_addresses = dict(zip(Types, zip(type_start_addresses,
                                        type_end_addresses)))
        else:
            type_addresses = dict(zip(Types, type_start_addresses))

        addresses.append(type_addresses)
        current_address = next_address

    return ADDRESS_TUPLE._make(addresses)
```

Quadruple processing and selection of operation

```
def play_note(lines, constants, directory_, inputs_):
    """ Entry point for orchestra """
    global directory
    directory = directory_
    global inputs
    inputs = inputs_
    global input_counter
    input_counter = 0

    memory['constant'] = constants

    # Clean the output
    for list_ in output:
        list_.clear()

    line_list = [line.split() for line in lines.split('\n')]

    current_quad_idx = 0
    while current_quad_idx < len(line_list):
        try:
            quad = line_list[current_quad_idx]
        except IndexError:
            # Finish when accessing non-existent quadruple (naive GOTO did it)
            return output

        try:
            operation = OPERATIONS[quad[0]]
        except KeyError:
            vm_result = handle_vm_function(quad, current_quad_idx)

            if vm_result is not None:
                current_quad_idx = vm_result
            else:
                current_quad_idx += 1
        except IndexError:
```

```
        # Empty operation (empty line) might only be found at the end
        return output
    else:
        handle_operation(operation, quad)
        current_quad_idx += 1

return output
```

2. MANUAL DE USUARIO

2.1. Symphony Quick Reference Guide

Symphony Program Structure

```
program symphony_welcome;
    print("Hello World"); // prints Hello World
```

Symphony Primitive Built-in Types

Type	Keyword
Boolean	bool
Character	char
Integer	int
Decimal	dec
Valueless	void

Variable Definition & Initialization in Symphony

```
int d;      // declaration of d
d = 3;   // initializing d
char x = 'x';        // the variable x has the value 'x'
```

Symphony Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. Symphony is rich in built-in operators and provides following type of operators:

Arithmetic Operators (+, -, \, *, ++, --)

Relational Operators (equals, >. <, >=, <=)
Logical Operators (and, or)
Assignment Operators (=)

Symphony Loop Type

while loop: Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

Symphony Decision Making

Symphony programming language provides following types of decision making statements.

Statement	Description
If statement	An if statement consists of a boolean expression followed by one or more statements.
If...else statement	An if statement can be followed by an optional else statement, which executes when the boolean expression is false.
If...elseif..else statement	An if statement can be followed by another elseif statement or optional else statement, which executes when the boolean expression is false.
nested if statements	You can use one if or else if statement inside another if or else if statement(s).

Symphony Functions

The general form of a Symphony function definition is as follows:

```
fun return_type function_name( parameter list )
{
   body of the function
}
```

- Return Type: A function may return a value. The return_type is the datatype of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the

return_type is the keyword void.

- Function Name: This is the actual name of the function. The function name and the parameter list together constitute the function signature.

- Parameters: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

- Function Body: The function body contains a collection of statements that define what the function does.

Numbers in Symphony

Following a simple example to show few of the mathematical operations on Symphony numbers:

```
program symphony_exponentation;
   int x;
   x = 2 ** 3;
   println(x);
   int modulo;
   modulo = 3 mod 2;
   println(modulo);
```

Symphony Arrays

Following is an example, which will show array declaration, assignment and accessing arrays in Symphony.

```
program symple_array;
str strings[10], strings2[7];
str greeting, farewell;

greeting = "hi";
farewell = "bye";
strings2[4 * 2 - 2] = "b";
strings[1] = "c";
strings[0] = "a";

println(greeting);
println(strings[0] + strings2[6] + strings[1]);
println(farewell);
```

Symphony Strings

Symphony provides following type of string representation:

```
program symphony_strings;
    str name, age, food;
    name = "Ali";
    age = "22";
    food = "roast beef";
    println(name);
    println(age);
    println(food);
```

Symphony Special Functions

Mathematical Functions	
sqrt	Square root of number
log	Natural logarithm of number
random	Random decimal number
ceil	Round up number
floor	Rounddown number
String Functions	
to_str	Convert value to string
length	Size of string
copy	Copy string to another string
get	Get character of the selected position in string
Musical Functions	
little_star	Play twinkle, twinkle, little star
C	Play C natural
D	Play D natural
E	Play E natural

F	Play F natural
G	Play G natural
A	Play A natural
B	Play B natural