Tecnológico de Monterrey

Documentación Final

Mat[e]

Natalia García Torres - A01191853
Miguel Angel Alvarado López - A01400121

Tabla de Contenidos

Descripción del Proyecto

Propósito, objetivos y alcance del proyecto.

Propósito del proyecto, ¿por qué Mat[e]?

Hoy en día, la cantidad de páginas, programas y tutoriales que existen para empezar a programar son innumerables, cada vez más gente se quiere involucrar y aprender a cómo programar, desde adultos que quieren generar ingresos al hacer una app, hasta infantes que sueñan con hacer un videojuego.

Los niños cada vez más jóvenes son expuestos a las diferentes tecnologías que existen en la actualidad y el fomentar su aprendizaje es una necesidad.

En México la cantidad de niños que son expuestos al internet o tecnologías es más de la mitad,

> "*Mientras que poco más de la mitad (53.1 por ciento) de los niños de entre 6 y 11 años señaló utilizar Internet con cierta regularidad, entre los adolescentes de 12 a 17 años la proporción alcanza el 85.5 por ciento…" [INEGI]*

y el exponerlos a conocimientos básicos de programación es una necesidad que puede ser cubierta de diferentes maneras.

En la actualidad ya existen varias webapps y aplicaciones que buscan enseñar a programar a niños, sin embargo la mayoría están enfocadas a la población inglés-hablante, y tomando en cuenta que solamente el *5% de la población habla inglés*, estaría dejando afuera a la mayoría de los niños del país. *[FINANCIERO]*

Buscando cubrir el mayor porcentaje de los niños en México (que solamente hablan español), buscamos crear un lenguaje de programación que sea interactivo, aprendible fácilmente y que funcione mediante programación en bloques de código.

Objetivo Principal y Área

El área al que está enfocado, es principalmente la educación, y su objetivo es el poder enseñar de la manera más sencilla posible, fundamentos básicos y lógica de programación.
Esto se logrará a través de un lenguaje en español *loosely typed*, que sea entendible y fácil de usar e implementar, para lograr que los niños se motiven a acercarse al área de la programación.

El público meta son niños hispano-hablantes que no tengan conocimientos sobre programación, y quieren hacer un acercamiento a esta área.

Alcance del proyecto

El proyecto busca satisfacer de manera satisfactoria y óptima las expectativas de la materia de “Diseño de Compiladores”, del semestre Enero-Mayo del 2018, haciendo un compilador funcional y que tenga un propósito.

Mat[e] es un compilador simple, que cumple con las funciones básicas de un lenguaje de programación. Busca soportar un lenguaje cumpla con el propósito establecido; enseñar a niños a poder programar. Funciona llamando bloques de código o funciones desde un “origen” (bloque principal de código), la forma en la que cumple el propósito es a través de una implementación que cuente con instrucciones simples para declarar variables, llamar funciones, que soporte de diferentes alcances, aritmética básica, ciclos, condiciones, e incluso recursividad, entre otras funciones básicas de un lenguaje de programación.

Mat[e] no es un lenguaje de programación que busque enseñar instrucciones complejas, por lo tanto no maneja estructuras de datos complejas y no es un lenguaje que cuente con la posibilidad de implementar clases y objetos.

Para lograr uno de los propósitos principales del proyecto, el que sea “amigable” y fácil de usar, adicional al lenguaje de programación, se creó una extensión para Visual Studio Code (VS Code), la cual consiste en un colorizer para el lenguaje dentro de VS Code y ya se encuentra en la tienda, es capaz de detectar cuando tienes un archivo con extensión “.mate” abierto, y te recomienda que la bajes.

Análisis de Requerimientos y Casos de Usos Principales

Requerimientos Funcionales

- El compilador será capaz de recibir un archivo de texto con código en Mate que compilara y ejecutará posteriormente.
- El compilador deberá de mostrar mensajes de error apropiados al usuario al momento de compilación
- El compilador deberá de mostrar mensajes de error apropiados al usuario durante la ejecución.
- El usuario podrá cargar un archivo al compilador.
- El compilador deberá de mostrar una salida de la ejecución del usuario.
- Durante la ejecución, el usuario podrá recibir salidas de texto.
- Durante la ejecución, el usuario podrá proporcionar información para el programa.

Requerimientos No Funcionales

- Los mensajes proporcionados por el compilador, no deberán de tardar más de un segundo en aparecer una vez compilado y ejecutado el código.
- Los mensajes proporcionados por la máquina virtual, no deberán de tardar más de un segundo en aparecer una vez compilado y ejecutado el código.
- Las salidas proporcionados por parte del sistema, de compilación, mensajes de posibles errores, o de fallas, deberán de ser claras de entender para el usuario.
- El compilador deberá de correr en cualquier computadora donde pueda correr la máquina virtual.
- El compilador deberá de tener instrucciones

Casos de Uso

Los Test Cases con los que se cuentan son los siguientes:

- Aritmetica
- Arreglos
- Crasher
- Factorial
- Fibonacci Recursión
- Matrices
- Parametros
- Fibonacci con Programación Dinámica

Se pueden ver más detalles sobre estos, así como pruebas estos en las secciones posteriores del documento.

Proceso Desarrollo

El proyecto fué desarrollado a lo largo de 10 semanas, teniendo de fecha el 25 de Febrero del 2018 , como la fecha de creación del repositorio con nombre “Mate”, y tuvo su primer commit el 28 del mismo mes.

El último commit fué realizado el dos de Mayo con un parche adicional.

Se utilizó Github como herramienta de control de versiones.
El repositorio (público) se puede encontrar aquí:

https://github.com/MikeAlvarado/Mate

Imagen corresponde al momento previo a la finalización del documento presente, pero no limita a que pueda haber posteriores previas a la entrega

Modificaciones aceptadas posteriormente, fueron propuestas durante las primeras iteraciones, haciendo cambio del entregable, la idea de que fuera un lenguaje sencillo, fácil de aprender y en español, fueron preservadas, sin embargo dejó de ser un lenguaje “visual” y se transformó en un lenguaje de “consola” orientado a objetos, que funciona con líneas/bloques de código.

Extensión de Mate

Para VS Code

Durante las últimas semanas de desarrollo se creó una extensión para Visual Studio Code (VS Code), la cual tiene función de un colorizer para el texto.
La extensión de optó por manejar de manera separada al compilador.

La extensión se completó y se subió a la tienda previa a la entrega final.

Reportes Semanales

Los reportes sobre el progreso semanales fueron puestos en la plataforma educativa del curso. Eventualmente, durante el desarrollo, se optó por agregar los archivos o reportes a GitHub.

#	Fecha	Reporte
1	Feb/Mar 26-2	**Análisis Léxico y Sintáctico** *El analizador léxico y sintáctico, funcionan y fueron corregidos de acuerdo a los diagramas propuestos, incluyendo el feedback que se dío de parte de la Ing. Elda.* *El test cases que corrimos funciona de manera esperada, muestran cuando no son aceptados, si el programa puede ser compilado, etc…* *De momento no detectamos que la gramática tenga ambigüedades, pues los diagramas que hicimos en la propuesta, fueron corregidos desde antes en la gramática.*
2	Mar 5-9	**Semántica Básica de Variables: Directorio de Procedimientos y Tabla de Variables** *En esta entrega se trabajó en corregir errores, extender funcionalidad y el análisis de semántica.* *Se hizo una actualización a el programa de prueba para verificar que el análisis semántico funciona de la manera esperada.* *Se agregó un error handler para el momento de revisar la compilación que funciona mediante un manejador de símbolos.* *Este puede:* ● *Verificar si la función está definida* ● *Verificar si variables están definidas para el alcance de la función*

		● *Verificar que no haya elementos duplicados* ● *Verificar que no haya identificadores duplicados* ● *Verificar que no haya variables duplicadas* ● *Verificar que no haya funciones duplicadas* *Además se agrego una lista que cuando marca el error, manda a consola mensajes sobre qué fué lo que falló, así mismo sobre qué variable/función es de donde proviene el problema.*
3	Mar 12-16	**Semántica Básica de Expresiones: Tabla de Consideraciones Semánticas (Cubo Semántico).** **Generación de código de Expresiones Aritméticas y estatutos secuenciales: Asignación, Lectura, etc…** *Se agregó la primera versión "cubo semántico" para revisar operaciones, asignaciones y lógica, que se relaciona con el trabajo de la vez pasada.* *El trabajo actual funciona con aritmética (sumar, restar, multiplicar y dividir) y da información sobre los tipos de posibles errores que puede tener.* *Aunque está en proceso, todavía no funcionan las asignaciones y todavía no se cuenta con la parte para revisar las operaciones lógicas.*
4	Mar 19-23	**Generación del Código de Estatutos Condicionales: Decisiones/Ciclos** Durante esta semana se tuvo revisión con el Ingeniero Ceballos, no se subió un reporte escrito a la plataforma. Se terminó posteriormente al reporte anterior los elementos faltantes de las expresiones aritméticas (asignaciones), así mismo como una actualización para verificar funcionamiento de las operaciones lógicas.
	Mar 26-30	**Semana Santa**
5	Abr 2-6	**Generación del Código de Funciones** Actualmente vamos "avanzados" de donde deberíamos de estar, desde la revisión con el Ingeniero Ceballos, ya teníamos avanzado el avance correspondiente para que nos lo checara. Aunque en ese momento faltaron algunas cosas y fallaron, se corrigieron posteriormente a la revisión. Se avanzó en la siguiente entrega.
6	Abr 9-13	**Mapa de Memoria de Ejecución para la Máquina Virtual.** **Máquina Virtual: Ejecución de Expresiones Aritméticas y Estatutos Secuenciales** Tuvimos una sesión extra sobre ponernos al corriente del trabajo de ambos. Se inició el desarrollo de la documentación final.

		En teoría deberíamos de estar trabajando en el mapa de memoria de ejecución de la máquina virtual, sus ejecuciones de expresiones aritméticas y estatutos secuenciales. *De primera instancia, terminamos la entrega anterior, correspondiente a la generación de código de funciones.* *Tuvimos un error en cuanto a las entregas y empezamos a trabajar en Generación de Código de Arreglos/Tipos Estructurados. Y aunque tuvimos progreso, nos dimos cuenta que no podíamos avanzar sin la máquina virtual.*
7	Abr 16-20	**Generación de Código de Arreglos/Tipos Estructurados.** **Máquina Virtual: Ejecución de Estatutos Condicionales** *"Además de completar la entrega anterior (Mapa de Memoria de Ejecución para la Máquina Virtual y Maquina Virtual: Ejecución de Expresiones Aritméticas y Estatutos Secuenciales), logramos completar la entrega de la semana (Generación de Código de Arreglos/Tipos Estructurados. Y Máquina Virtual: Ejecución de Estatutos Condicionales).* *Actualmente se pueden realizar if, whiles, e incluso funcionan con arrays. Estamos al corriente, por lo cual quiere decir que solamente faltan realizar pruebas del código.* *Se avanzó en la documentación final."*
8	Abr 23-27	**1era Versión de la documentación Final** **Generación de código y Máquina Virtual para una parte de la aplicación en particular.** *"Para esta entrega se debería de haber entregado 1era Versión de la documentación Final Generación de código y Máquina Virtual para una parte de la aplicación en particular. El compilador como tal, ya se encuentra terminado, se desarrollaron pruebas para el mismo y se corrigieron detalles menores para que todo funcionara.* *Actualmente la documentación está bastante avanzada, con más de 17 páginas.* *Se está trabajando de igual manera en la generación de una extensión para el lenguaje que cuente con un colorizer para VS Code y probablemente cuente con snippets de código."*
9	Abr/May 30-3	**Entrega Final del Proyecto**

Conclusiones Personales

Las conclusiones fueron escritas previas a la entrega, una vez marcado como completado el proyecto.

Lo que me llevo del proyecto es la satisfacción de haber logrado hacer algo tan complejo desde cero. En esta clase realmente sentí que el proyecto incorpora muchos de los conocimientos que había obtenido previamente.
Fue relativamente complicado manejar las versiones a través de GitHub en un proyecto "grande" donde se modifican tantos archivos al mismo tiempo.
Pude aprovechar y aumentar las capacidades profesionales que tengo en su totalidad. Estoy contenta con el resultado y de haber trabajado en un proyecto con un compañero donde el dividir el trabajo también era un reto.
Creo que llevar esta materia ya apunto de graduarme es el "sello" real de que estoy lista como una ingeniera de software.

_____

Natalia García Torres

La clase de compiladores fué todo un reto de diferentes maneras, desde la teoría, exámenes, el desarrollo y ahora la documentación, fué una etapa bastante diferente.
Sin duda, y aunque mi parte durante la programación fue menor que la de mi compañera, este ha sido el proyecto más largo que he desarrollado para la escuela.
Estoy impresionado con no solamente los elementos que tiene un lenguaje de programación "sencillo" como el nuestro, pero todo lo que tiene que hacer internamente los diferentes compiladores y lenguajes que conocemos para hacer cosas que consideramos sencillas, como mandar a llamar una función hasta algo más complicado como recursividad.
Me agradó bastante trabajar con un compañero durante todo el desarrollo del proyecto, en lo personal estoy bastante agradecido por todo el apoyo y ayuda que me proporcionó.
Me fué bastante interesante estar apoyando a alguien durante el desarrollo y estoy muy feliz con el trabajo final.

_____

Miguel Angel Alvarado López

Descripción del Lenguaje

Nombre del Lenguaje

Mat[e]

Descripción General

Mat[e] pretende ser un lenguaje de programación orientado a objetos, se maneja a través de una consola, su principal característica y propósito es que sea fácil de aprender y usar por personas jóvenes hispanohablantes. Tiene la característica de ser muy amigable para el usuario y funciona a través de líneas o bloques de código.

Características del Lenguaje

- La estructura consiste en líneas de código.
- La interfaz de usuario es proporcionada por el shell de comandos del sistema operativo.
- El shell espera que proporciones un archivo de texto.
- La salida será mostrada en el shell.
- Información sobre errores de compilación/mensajes para el usuario, serán proporcionados en el shell.

El lenguaje soporta tipos de datos primitivos. Y para que sean fácil de entender y que sirvan como base para aprender, se mantuvieron los tipos

- String
- Int
- Float
- Bool

Se cuenta con arreglos, y es la forma en la que se pueden tener colecciones de números

- Array

Además de que puede identificar si las variables no han sido declarados o son de un tipo inválido

- Undefined
- Invalid

Código	Tipo
0	STRING
1	INT
2	FLOAT
3	BOOL
4	ARRAY
5	INVALID
6	UNDEFINED

Listado de Errores

Mate maneja los errores a través de un handler, un sistema de validación. Los errores causan que el compilador se detenga por completo, y que se despliegue un mensaje correspondiente.

Errores Compilación

- Nombre del bloque previamente declarado
- Variable previamente declarada
- Palabra Reservada

Errores RunTime

- Palabra Reservada
- Operación Inválida
    - Operador Inválido
    - Operando Inválido
    - Operandos Inválidos
- Límites (Índices fuera de los límites)
- Memoria Insuficiente
- Variables Inexistente

Descripción Compilador

Herramientas

El compilador fué desarrollado en dos computadoras con sistema operativo Mac OS High Sierra (10.13.x) en Ruby 2.5.
Adicionalmente fué utilizado RACC, una herramienta para generar el parser, y Rexical, un analizador léxico, ambos OpenSource.

Análisis Léxico

Nombre	Expresión Regular
program	“programa”
function	“funcion”
origin	“origen”
var	“var”
if	“si”
no	“no”
but	“pero”
return	“regresa”
while	“mientras”
right_bracket	“]“
left_bracket	“[”
right_parenthesis	“)“
left_parenthesis	“(”
op_not	“!”
op_not_equal	“!=”
op_equal	“==”

op_greater	“>”
op_greater_equal	“>=”
op_less	“<”
op_less_equal	“<=”
op_add	“+”
op_subtract	“-”
op_multiply	“*”
op_divide	“/”
op_and	“&&”
op_or	“||”
op_assignation	“=”
comma	“,”
semicolon	“;”
colon	“:”
cst_integer	[0-9][0-9]*
cst_decimal	([0-9]*[.])?[0-9][0-9]*
cst_string	\"(\\. | [^\”])*\”
cst_boolean	“cierto” | ”falso”
id	[a-zA-Z_][_a-zA-Z0-9]*
semicolon	“;”
comma	“,”
read	“lee”
write	“escribe”
element_size	“talla”

Análisis Sintáctico

La gramática libre de contexto cuenta con las siguientes reglas.

Regla Sintáctica	Gramática Libre de Contexto
program	**program_def** L_BRACKET **functions** R_BRACKET
program_def	**PROGRAM ID**
functions	**function _functions**
_functions	**function _functions** | Ɛ
function	**FUNCTION _function params block**
_function	**ORIGIN** | **ID**
params	L_PAREN **_params** R_PAREN | Ɛ
_params	**ID more_params** | Ɛ
more_params	COMMA **ID more_params** | Ɛ
block	L_BRACKET **statements** R_BRACKET
statements	**simple_statement** SEMICOLON | if_else | while
simple_statement	**var_declaration** | **var_assign** | **function_call** | return
var_declaration	**VAR ID more_declarations**
more_declarations	**_more_declarations**

	| op_assign **_more_declarations**
_more_declarations	COMMA **ID more_declarations** | ε
var_assign	**var_value** op_assign
op_assign	OP_ASSIGN **expression**
constant	CST_STR | CST_INT | CST_DEC | **cst_bool** | NIL
cst_bool	TRUE | FALSE
var_value	ID **array_access**
array	L_SQ_BRACKET **values** R_SQ_BRACKET
values	constant _values | ε
_values	COMMA **constant _values** | ε
array_access	L_SQ_BRACKET **expression** R_SQ_BRACKET | ε
function_call	**built_in_function_call** | **custom_function_call**
built_in_function_call	WRITE L_PAREN **expression** R_PAREN | READ L_PAREN R_PAREN
custom_function_call	**call_name** L_PAREN **call_params** R_PAREN
call_name	ID
call_params	**expression _call_params**
_call_params	COMMA **expression _call_params** | ε
return	RETURN **expression**

while	WHILE L_PAREN **expression** R_PAREN **block**
if_else	IF L_PAREN **expression** R_PAREN **block _if_else**
_if_else	BUT IF NO **block** | Ɛ
expression	not exp **_expression**
_expression	and_or **expression** | Ɛ
not	OP_NOT | Ɛ
exp	item **_exp**
_exp	ogic_op **exp** | Ɛ
logic_op	OP_GREATER | OP_LESS | OP_EQUAL | OP_GREATER_EQUAL | OP_LESS_EQUAL | OP_NOT_EQUAL
and_or	OP_OR | OP_AND
item	term **_item**
_item	**add_subtract item** | Ɛ
add_subtract	OP_ADD | OP_SUBTRACT
term	factor **_term**
_term	**multiply_divide term** | Ɛ
multiply_divide	OP_MULTIPLY | OP_DIVIDE | OP_MOD
factor	L_PAREN **expression** R_PAREN

	| **_add_subtract** value
_add_subtract	**add_subtract** | ε
value	**constant** | **var_value** | **array** | **function_call**

Descripción de Generación de Código Intermedio y Análisis Semántico.

Códigos Operación

La forma en la que trabajamos la semántica y su uso (con el cubo semántico explicado más adelante) para los cuádruplos es con tokens, es a través de dos diferentes módulos, uno para los operadores (unarios y binarios), y otro para los saltos.

Operation Tokens

Binarios (0..12)
Unarios (13..16)

Código	Palabra	Símbolo (match)
0	EQUAL	'=='
1	NOT_EQUAL	'!='
2	LESS_EQUAL	'<='
3	GREATER_EQUAL	'>='
4	LESS	'<'
5	GREATER	'>'
6	AND	'&&'
7	OR	'||'
8	MOD	'%'
9	MULTIPLY	'*'

10	DIVIDE	‘/’
11	ADD	‘+’
12	SUBSTRACT	‘-’
13	NOT	‘!’
14	ASSIGN	‘=’

Jump Tokens

Código	Palabra
15	READ
16	WRITE
17	GOSUB
18	GOTO
19	GOTOF
20	EOF
21	EOP
22	ERA
23	PARAM
24	RETURN
25	SOF

Direcciones Virtuales (Límites Memoria)

Una de nuestras principales features del compilador es que nuestro lenguaje es **“loosely typed”** y es debido a esto que no al momento de crear las variables, no podemos saber cuánto espacio van a requerir, optamos por solamente dividir la memoria en variables locales y temporales. Teniendo como límite de variables 5,000 y un Frame no mayor a 500

Tipo Memoria	Inicio
Local **(LOCAL_START_ADDR)**	0
Temporal **(TEMP_START_ADDR)**	5000

Diagramas de Sintaxis

program
function
block 1. Genera el “End-Proc”.
statement

simple_statement NA
while 1. Verifica que la expresión evalúa a un tipo booleano, y verifica que ya no haya saltos pendientes
if_else
var_declaration
var_assign

function_call 1. Se agrega id a la pila de function_calls, se hace un ERA 2. Se hace un cuádruplo para cada parámetro 3. Se verifica que los parámetros concuerden 4. Se hace el GOSUB y se especifica dónde poner el valor de retorno (si hay)
return 1. Cuádruplo de RETURN con el operando que regresa
condition_block 1. Hace push del punto inicial a pila de saltos pendiente
var_value
expression

1. Agrega un operador “Not” al stack
2. Verifica que lo que regresa sea booleano
3. Agrega (“And” | “Or”)

exp

1. Push a operador relacional indicado al stack
2. Verifica que no haya elementos restantes en el stack, y si hay, verifica el pasado para hacerlo.

Item

1. Se agrega el operador “+” ó “-” a la pila de operadores

2. Se evalúa la expresión binaria
term 1. Se agrega el operador “+” ó “-” a la pila de operadores 2. Se evalúa la operación binaria
factor 1. Se agrega el operador “+” ó “-” a la pila de operadores 2. Se mete “value” a la pila de operandos y si agrega el +- si es un signo 3. Valida si es una expresión válida d

logical_op	value
constant 1. Agregar el tipo de constante	array 1. Agregar el valor a la lista

Características Semánticas

- La palabra “origen” es la que decide cúal es la función inicial (similar a lo que hace “main”).
- Las variables tienen que ser declaradas antes de ser usadas
- Los nombres (IDs) de las variables, no pueden ser cambiados.
- Un arreglo solamente puede contener un tipo de elemento.
- Solamente las operaciones mencionadas a continuación son aceptadas por el lenguaje.

Tipos de Operadores

Se manejan los operadores por tres tipos: aritméticos, comunes y lógicos.
Separados de la siguiente manera.

Tipo	Palabra	Símbolo (match)
common	EQUAL	'=='
common	NOT_EQUAL	'!='
common	AND	'&&'
common	OR	'||'
logic_ops	LESS_EQUAL	'<='
logic_ops	GREATER_EQUAL	'>='
logic_ops	LESS	'<'
logic_ops	GREATER	'>'
arithmetic_ops	MOD	'%'
arithmetic_ops	MULTIPLY	'*'
arithmetic_ops	DIVIDE	'/'
arithmetic_ops	ADD	'+'
arithmetic_ops	SUBSTRACT	'-'

Operaciones Aceptadas

Operando	Operador	Operando	Resultado
STRING	+	STRING | INT | FLOAT | BOOL | ARRAY	STRING

STRING	+	UNDEFINED	UNDEFINED
STRING	common	INT | FLOAT | BOOL | ARRAY | Invalid	BOOL | Undefined
STRING	logic_ops	STRING | Undefined	BOOL
INT	+	STRING	STRING
INT	common	STRING	BOOL | Undefined
INT	arithmetic_ops	INT	INT
INT	arithmetic_ops	FLOAT	FLOAT
INT	arithmetic_ops	Undefined	Undefined
INT	logic_ops	INT | FLOAT | Undefined	BOOL
INT	common	BOOL | ARRAY | Invalid	BOOL | Undefined
FLOAT	+	STRING	STRING
FLOAT	common	STRING BOOL | ARRAY | Invalid	BOOL | Undefined
FLOAT	arithmetic_ops	INT | FLOAT	FLOAT
FLOAT	logic_ops	INT | FLOAT | Undefined	BOOL
FLOAT	arithmetic_ops	Undefined	Undefined
BOOL	+	STRING	STRING
BOOL	common	INT	BOOL

		| FLOAT | BOOL | ARRAY | Invalid | Undefined	| Undefined
ARRAY	+	STRING	STRING
ARRAY	+ | -	ARRAY	ARRAY
ARRAY	+ | -	Undefined	Undefined
ARRAY	common	INT | FLOAT | BOOL | ARRAY | Invalid | Undefined	BOOL | Undefined
INVALID	common	INT | FLOAT | BOOL | ARRAY | Invalid | Undefined	BOOL | Undefined
UNDEFINED	+	STRING	Undefined
UNDEFINED	+ | -	ARRAY | Undefined	Undefined
UNDEFINED	common	STRING | BOOL | ARRAY | Invalid	BOOL | Undefined
UNDEFINED	logic_ops	INT | FLOAT	**Undefined**
UNDEFINED	arithmetic_ops	INT | FLOAT | Undefined	**Undefined**

Administración de Memoria Usado en la Compilación

La forma en la que se maneja la memoria es a través de un manager, cuenta con los diferentes elementos.

Memory::Manager

Elemento	Descripción
local_counter	Número de direcciones usadas para las variables locales. Se le asigna este valor como dirección de memoria a una variable local a menos a que haya direcciones liberadas para locales
memory	Hash Key: nombre de variable Value: Memory::Entry o Symbols::Var
temp_counter	Número de direcciones usadas para las variables temporales. Se le asigna este valor como dirección de memoria a una variable temporal a menos a que haya direcciones liberadas para temporales
temp_memory	Arreglo de entradas (Memory::Entry o Symbols::Var) de variabels temporales
deallocated	Direcciones de variables locales liberadas
temp_deallocated	Direcciones de variables temporales liberadas

Memory::Entry

Elemento	Descripción
addr	Número de dirección de memoria de la variable
is_temp	Booleano para saber si la variable es temporal o local

Symbols::Var (hereda de Memory::Entry)

Elemento	Descripción
name	Nombre de la variable

Máquina Virtual

La máquina virtual tiene una memoria (RuntimeMemory) que maneja frames. Hay un frame por cada función.
Cuando se corre la máquina virtual se inicializa la memoria con los datos de la función origen

VM::RuntimeMemoy

Elemento	Descripción
call_stack	Stack de frames
var_count	Número de variables en el programa

VM::Frame

Elemento	Descripción
dispatcher	Frame que llamó a la función del frame actual
size	Número de variables locales que usa la función
name	Nombre de la función
params	Lista de parámetros de la función
local	Lista de variables locales
temp	Lista de variables temporales

Pruebas

hola_mundo.mate

Aritmetica con arreglos que pasamos como funciones
aritmetica.mate

sort.mate
El usuario introduce un arreglo y el programa lo ordena con bubble sort

fibonacci_recursion.mate
Implementación de fibonacci recursivo

matrices.mate

prog_dinamica_fibonacci.mate

Implementación

El código completo está documentado y referenciado apropiadamente el el GitHub del proyecto. Todas las funciones y módulos cuentan con comentarios propios a su funcionalidad y lo que hacen.

Comentarios

Los comentarios presentes en ruby están marcados e inician con un “#”.

Elementos Principales

Escoger los elementos “principales” fué bastante complicado, ya que cualquier elemento es necesario para la funcionalidad completa.
Se escogieron aquellas que se relacionan con otras, o los “helpers”

Nombre: class Quadruple
Ubicación: src/ir/quadruple.rb
Descripción: class Quadruples es uno de los elementos principales del lenguaje, consideramos que es así pues es donde se generan y leen las instrucciones que se van a ejecutar.

Nombre: module Ir
Ubicación: src/ir/quadruples.rb
Descripción: Este módulo es donde se encuentra el generador de cuadruplos, aquí, además de contar con la clase de cuadruplos, que es donde se inicializan las pilas que se van a manejar en los cuádruplos.

Nombre: memory
Ubicación: src/memory/manager.rb
Descripción: Manager es donde se maneja la memoria de compilación. En esta memoria es donde se cargan las variables. Al solamente tener tipo de memoria temporal y local, el manejador es bastante especial, ya que no está separada por más limitantes

Nombre: Function_Helper
Ubicación: src/parser/helper.rb
Descripción: Helper es la clase que contiene los métodos que llama el parser. Junta el módulo representación intermedia (IR), quien es quien maneja el estado de los cuádruplos, así mismo como cómo llamar a las funciones necesarias.

Nombre: Validator
Ubicación: src/validators/validate.rb
Descripción: Es la clase auxiliar que ayuda a desplegar los errores. Durante el run y el compile time. Básicamente es quien revisa si el texto a alimentar y la ejecución es adecuada

Referencias

- Flanagan, D., & Matsumoto, Y. (2008). The Ruby programming language. Sebastopol, CA: O'Reilly.

- Estadísticas a propósito del día mundial del internet. Retrieved from http://www.inegi.org.mx/saladeprensa/aproposito/2017/internet2017_Nal.pdf

- En México sólo 5% de la población habla inglés: IMCO. Retrieved from http://www.elfinanciero.com.mx/economia/en-mexico-solo-de-la-poblacion-habla-ingles-imco.html