Tecnológico de Monterrey

Campus Monterrey

Diseño de Compiladores

# MIRI

# Lenguaje de programación

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |

Ricardo Licea Mata Miguel Bazán Aviña

A01280892 A01281010

Profesores:

* Ing. Elda Quiroga
* Dr. Héctor Ceballos

## Descripción del proyecto

#### Propósito, objetivos y alcance

##### Propósito

Dentro del plan de estudios de la carrera de Ingeniería en Tecnologías Computacionales, es fundamental la comprensión del proceso de desarrollo de un compilador. Gracias al curso de Desarrollo de Compiladores, ofrecida en el Tecnológico de Monterrey, se ha podido estudiar el proceso de desarrollo de un compilador, el cual termina el curso con el desarrollo e implementación de un compilador. De esta manera, el propósito de nuestro proyecto, en conjunto con los temas vistos en clase, es tener una mejor percepción del uso de la memoria, del poder computacional que se ocupa para correr un programa; y, con esto, poder escribir código de una manera más eficiente y rápida.

##### Objetivo

Nuestro proyecto, tiene por nombre MIRI, el cual, corresponde a las siglas de los miembros del equipo, tiene como características varios puntos:

* Ofrecer una sintaxis casi obvia
* Tener representaciones gráficas como salidas
* Ser un lenguaje, aunque tedioso, muy intuitivo

Dichas características, fueron pensadas con el objetivo de que MIRI, sea utilizado por gente interesada en el mundo de la codificación, sin embargo que al momento de utilizarse, no tengan tanto fundamento en las bases de la programación. De esta manera, la comprensión y entendimiento acerca del proceso de desarrollo de software puede hacer que los programadores, tengan un arranque más eficiente en los estudios de desarrollo de software.

##### Alcance

El desarrollo de nuestro proyecto, comprende las tres fases de Análisis del desarrollo de un compilador: Léxico, Sintaxis, y Semántico. Posteriormente, apoyados de una máquina virtual, se hará la parte final del proyecto, abusando de la máquina virtual para reservar, generar y utilizar los espacios de memoria correspondientes, así como el utilizo de código intermedio, en una estructura definida por cuadruplos, para poder generar código de alto nivel.

#### Análisis de Requerimientos y Casos de uso generales

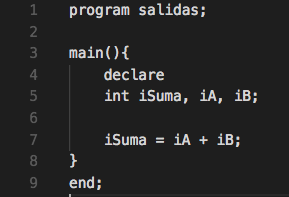
##### Análisis de requerimientos

* + 1. R.1: Deberá poder definir variables locales.
    2. R.2: Deberá poder definir funciones con y sin parámetros de entrada
    3. R.3: Deberá poder hacer operaciones aritméticas (suma, resta, multiplicación, división)
    4. R.4: Deberán poder crear arreglos de 1 y 2 dimensiones
    5. R.6: Deberá poder dibujar figuras geométricas (círculo, línea, cuadrado, triángulo)
    6. R.7: Deberá poder definir variables globales
    7. R.8: Deberá poder definir ciclos (while, do-while)
    8. R.9: Deberá poder definir los tipos Int, String, Bool, Float
    9. R.10: Deberá poder definir funciones con y sin valores de retorno
    10. R.11: Se deberá poder hacer asignaciones a variables.
    11. R.12: Se deberá poder crear condicionales (IF – ELSE)
    12. R.13: Se deberá poder escribir en pantalla (print)
    13. R.14: Se deberá poder leer datos del teclado (read)
    14. R.15: Se deberá poder tener expresiones con desigualdades (<,>, <=,>=,==, !=)

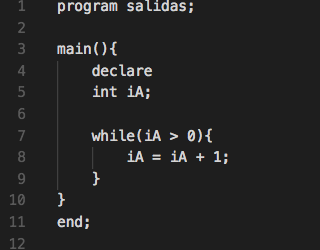
##### Casos de uso

Durante el desarrollo de proyecto se fue probando el código, con pruebas que puedan cubrir todos los requerimientos del compilador:

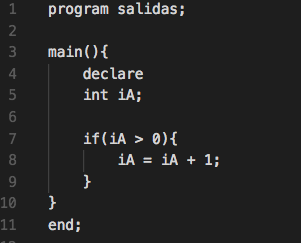
* Expresiones aritméticas básicas



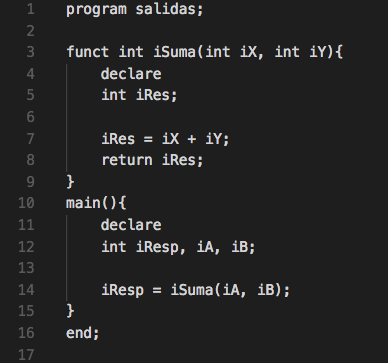
* Ciclos

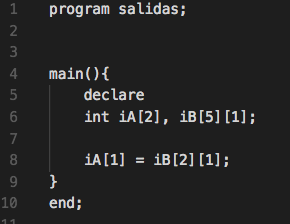


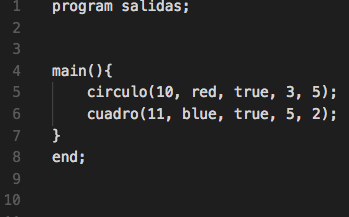
* Condicionales



* Llamada a funciones



* Acceso a casillas de arreglos de 1 y 2 dimensiones
* Llamada a funciones con salidas gráficas



#### Descripción de los principales test cases

De la misma manera, se desarrollaron las siguientes pruebas para cumplir con los requerimientos del proyecto:

* Fibonnaci: Función que calcula la sucesión de Fibonnaci. Deberá cumplir tanto cíclica, como recursivamente
* Factorial: Función que calcula el número factorial de un número. Deberá cumplir tanto cíclica, como recursivamente
* Accesar a alguna casilla en especifica de un arreglo
* Multiplicación de matrices

#### Proceso seguido

El desarrollo del proyecto consistió en 5 partes:

* Análisis léxico
* Análisis sintáctico
* Análisis semántico
* Generación de código intermedio
* Implementación de máquina virtual y salidas gráficas

Las primeras tres partes, pertenecientes al análisis, se desarrollaron en la primer parte del semestre. Originalmente, se iba a utilizar ANTLR v.4 para generar el análisis léxico. Sin embargo, se opto por desarrollar el análisis usando PLY. La generación de código intermedio parte a raíz del análisis sintáctico y la implementación de la maquina virtual fue hecha en Python. Nos dividimos tareas durante la fase de análisis para poder generar el código intermedio y una vez terminada, enfocarnos en diferentes contextos de la máquina virtual.

### Descripción del lenguaje

#### Nombre del Lenguaje

* + - 1. Miri

#### Descripción de las principales características de lenguajes

* + - 1. El lenguaje cuenta con funciones básicas de cualquier lenguaje de programaciones tales como condicionales, ciclos, arreglos, definición de variables, definición de funciones, lectura y escritura. Además el lenguaje es capaz de crear figuras geométricas básicas tales como circulo, línea, cuadrado y triángulo.

#### Listado de los errores

##### Compilación

Estos son los errores marcados al usuario:

* Error de sintaxis: cuando el programa no coincide con la sintaxis del programa. Muestra el Token incorrecto y la línea en la que se encuentre
* Variable no declarada: cuando se utiliza una variable que no se encuentra en el directorio de procedimientos

### Descripción del compilador

#### Equipo de cómputo, lenguaje y utilerías

##### Equipo y ambiente de trabajo

* + - * 1. MacBook Pro 2017 y MacBook Air 2016
        2. MacOS High Sierra

##### Lenguaje de desarrollo

* + - * 1. Python

##### Utilerias de desarrollo

* + - * 1. Visual Studio Code
        2. Ply
        3. Turtle

#### Análisis Léxico

##### Expresiones Regulares

|  |  |  |
| --- | --- | --- |
| Expresión Regular | Token | Explicación |
| '[a-zA-Z\_][a-zA-Z0-9\_]\*' | ID | Cualquie cadena de caractéres |
| '\n+' | New Line | Salto de línea |
| 'false|true' | Bool | Valor booleano |
| '\#.\*' | Comentarios | Para reconocer comentarios |
| r'=' | Assignment | Asignación |
| r',' | Comma | Coma |
| r'\.' | Dot | Punto |
| r':' | Colon | Dos puntos |
| r';' | SemiColon | Punto y coma |
| r'\[' | LeftBracket | Llave cuadrada izquierda |
| r'\]' | Rightracket | Llave cuadrada derecha |
| r'\(' | LeftPar | Paréntesis izquierdo |
| r'\)' | RightPar | Paréntesis derecho |
| r'{' | LeftKey | Llave izquierda |
| r'}' | RightKey | Llave derecha |
| r'"' | Quotes | Comillas |
| r'\+' | Plus | Mas |
| r'\-' | Minus | Menos |
| r'\\*' | Times | Por |
| r'/' | Divide | Dividir |
| r'>' | Greater | Mayor que |
| r'<' | Less | Menor que |
| r'==' | Equal | igual |
| r'!=' | Not equal | Diferente de |
| r'>=' | Greater than | Mayor igual que |
| r'<=' | Less than | Menor igual que |
| r'[\+,-]?\d+' | Integer | Número entero |
| r'[+,-]?[0-9]+\.[0-9]+((E|e)[+,-]?[0-9]+)?' | Float | Número flotante |

##### Tokens

Los tokens, son todas las palabras reservadas por el lenguaje. A continuación enlistamos todas:

* END
* CIRCLE
* LINE
* TRIANGLE
* SQUARE
* BOOL
* RETURN
* INTEGER
* DECLARE
* INT
* FLOAT
* STRING
* BOOL
* VOID
* MAIN
* IF
* ELSE
* DO
* WHILE
* WRITE
* READ
* FUNCTION
* PROGRAM
* ID
* ASSGN
* TYPE
* COMMA
* DOT
* CYCLE
* COLON
* SEMICOLON
* FLOATNUMB
* LEFTBRACK
* RIGHTBRACK
* LEFTPAR
* RIGHTPAR
* LEFTKEY
* RIGHTKEY
* QUOTE
* SUM
* OR
* AND
* MINUS
* MULTP
* DIVIDE
* GRTR LESS
* EQ
* NOTEQ
* GRTREQ
* LESSEQ
* NUMBER
* newline
* SPACE

#### Análisis de Sintaxis

Nuestro analizador sintáctico, consiste en el archivo “parser.py”, en el cual esta definida la gramática formal del lenguaje, basándose en los tokens definidos en el analizador léxico.

##### Gramáticas Formales

|  |
| --- |
| Grammar |
|  |  |
|  | Rule 0 S' -> program |
|  | Rule 1 program -> PROGRAM ID SEMICOLON program2 cuerpo END SEMICOLON |
|  | Rule 2 program2 -> declare program3 |
|  | Rule 3 program2 -> empty |
|  | Rule 4 declare -> DECLARE declareRecursivo |
|  | Rule 5 declare -> empty |
|  | Rule 6 declareRecursivo -> type ID assignmentDecl declare2 declare3 SEMICOLON     declareRecursivo |
|  | Rule 7 declare2 -> array |
|  | Rule 8  declare3 -> COMMA ID declare3 |
|  | Rule 9  program3 -> funct program3 |
|  | Rule 10 funct -> FUNCTION type ID LEFTPAR funct2 RIGHTPAR LEFTKEY est     functReturn RIGHTKEY |
|  | Rule 11 functReturn -> RETURN NUMBER SEMICOLON |
|  | Rule 12 functReturn -> RETURN ID SEMICOLON |
|  |  |
|  | Rule 13 funct2 -> type ID funct3 |
|  | Rule 14 funct3 -> COMMA type ID funct3 |
|  | Rule 15 funct2 -> empty |
|  | Rule 16 funct3 -> empty |
|  | Rule 17 program3 -> empty |
|  |  |
|  | Rule 18 declareRecursivo -> empty |
|  | Rule 19 declare3 -> empty |
|  | Rule 20 array -> LEFTBRACK exp RIGHTBRACK array |
|  | Rule 21 array -> empty |
|  | Rule 22 type -> type2 |
|  | Rule 23 type2 -> INT |
|  | Rule 24 type2 -> FLOAT |
|  | Rule 25 type2 -> STRING |
|  | Rule 26 type2 -> BOOL |
|  | Rule 27 type2 -> VOID |
|  | Rule 28 cuerpo -> MAIN LEFTPAR RIGHTPAR LEFTKEY est RIGHTKEY |
|  | Rule 29 est -> conditional est |
|  | Rule 30 est -> declareLocal est |
|  | Rule 31 est -> cycles est |
|  | Rule 32 est -> input est |
|  | Rule 33 est -> output est |
|  | Rule 34 est -> assignment est |
|  | Rule 35 est -> llamadaAFunct est |
|  | Rule 36 est -> empty |
|  | Rule 37 llamadaAFunct -> ID LEFTPAR llamadaAFunct RIGHTPAR |
|  |
|  |
|  | Rule 38 llamadaAFunct -> empty |
|  | Rule 39 llamadaAFunct2 -> exp llamadaAFunct3 |
|  | Rule 40 llamadaAFunct3 -> COMMA llamadaAFunct2 |
|  | Rule 41 llamadaAFunct3 -> empty |
|  | Rule 42 llamadaAFunct2 -> empty |
|  | Rule 43 declareLocal -> DECLARE declareRecursivoLocal |
|  | Rule 44 declareRecursivoLocal -> type ID assignmentDecl declare2Local     declare3Local SEMICOLON declareRecursivoLocal |
|  | Rule 45 assignmentDecl -> ASSGN exp |
|  | Rule 46 number -> INTEGER |
|  | Rule 47 number -> FLOAT |
|  | Rule 48 number -> empty |
|  | Rule 49 assignmentDecl -> empty |
|  | Rule 50 declare2Local -> array |
|  | Rule 51 declare3Local -> COMMA ID assignmentDecl declare3Local |
|  | Rule 52 declareRecursivoLocal -> empty |
|  | Rule 53 declareLocal -> empty |
|  | Rule 54 declare2Local -> empty |
|  | Rule 55 declare3Local -> empty |
|  |  |
|  | Rule 56 assignment -> ID ASSGN exp SEMICOLON |
|  | Rule 57 assignment -> ID ASSGN llamadaAFunct SEMICOLON |
|  | Rule 58 assignment -> empty |
|  | Rule 59 conditional -> IF LEFTPAR conditional2 RIGHTPAR LEFTKEY est RIGHTKEY     conditionalElse |
|  |
|  |
|  | Rule 60 conditionalElse -> ELSE LEFTKEY est RIGHTKEY |
|  |  |
|  | Rule 61 conditionalElse -> empty |
|  | Rule 62 conditional2 -> exp conditional2 |
|  | Rule 63 conditional2 -> empty |
|  | Rule 64 cycles -> while |
|  | Rule 65 cycles -> empty |
|  | Rule 66 cycles -> do-while |
|  | Rule 67 do-while -> DO LEFTKEY est RIGHTKEY WHILE LEFTPAR while2 RIGHTPAR |
|  | Rule 68 while -> WHILE LEFTPAR while2 RIGHTPAR LEFTKEY est RIGHTKEY |
|  | Rule 69 while2 -> exp while2 |
|  | Rule 70 while2 -> empty |
|  | Rule 71 exp -> ID exp2 |
|  | Rule 72 exp -> LEFTPAR exp RIGHTPAR exp2 |
|  | Rule 73 exp -> number exp2 |
|  | Rule 74 exp -> empty |
|  | Rule 75 exp2 -> LESS exp |
|  | Rule 76 exp2 -> GRTR exp |
|  | Rule 77 exp2 -> EQ exp |
|  | Rule 78 exp2 -> NOTEQ exp |
|  | Rule 79 exp2 -> AND exp |
|  | Rule 80 exp2 -> OR exp |
|  | Rule 81 exp2 -> SUM exp |
|  |
|  | Rule 82 exp2 -> MINUS exp |
|  | Rule 83 exp2 -> MULTP exp |
|  | Rule 84 exp2 -> DIVIDE exp |
|  | Rule 85 exp2 -> empty |
|  | Rule 86 output -> WRITE LEFTPAR output2 RIGHTPAR SEMICOLON |
|  | Rule 87 output2 -> ID output2 |
|  | Rule 88 output2 -> QUOTE ID QUOTE output2 |
|  | Rule 89 output2 -> empty |
|  | Rule 90 input -> READ LEFTPAR ID RIGHTPAR SEMICOLON |
|  | Rule 91 empty -> <empty> |
|  | Rule 92 circulo -> LEFTPAR INTEGER COMMA STRING COMMA BOOL COMMA INTEGER COMMA INTEGER RIGHTPAR SEMICOLON |
|  | Rule 93 cuadro -> LEFTPAR INTEGER COMMA STRING COMMA BOOL COMMA INTEGER COMMA INTEGER RIGHTPAR SEMICOLON |
|  | Rule 94 triangulo -> LEFTPAR INTEGER COMMA INTEGER COMMA INTEGER COMMA STRING COMMA BOOL COMMA INTEGER COMMA INTEGER COMMA INTEGER RIGHTPAR SEMICOLON |
|  | Rule 95 linea -> LEFTPAR INTEGER COMMA STRING COMMA INTEGER COMMA INTEGER COMMA INTEGER COMMA INTEGER RIGHTPAR SEMICOLON |
|  | Rule 96 rectangulo -> LEFTPAR INTEGER COMMA INTEGER COMMA STRING COMMA BOOL COMMA INTEGER COMMA INTEGER COMMA INTEGER RIGHTPAR SEMICOLON |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

#### Generación de código Intermedio y análisis semántico

Dentro del análisis semántico, primero especificamos en el cubo semántico las posibles combinaciones dependiendo los tipos y los operandos. Posteriormente, proseguimos a la generación de código intermedio de todos los estatutos. El tipo de código intermedio que utilizamos es de Cuádruplos, utilizando la siguiente sintaxis:

(operador, operandoIzq, operandoDer, dirTempRes )

Teniendo el siguiente significado:

* Operador: El operador de la operación. Pudiera ser un operador aritmético, lógico o en su caso, un código de operación
* operandoIzq: El lado izquierdo de una operación
* operandoDer: El lado derecho de una operación
* dirTempRes: Dirección temporal para guardar el resultado del cuádruplo

##### Códigos de operación y direcciones virtuales

A continuación, enlistamos los códigos de operación

|  |  |  |
| --- | --- | --- |
| Código de operación | Explicación | Ejemplo |
| GOTO | Cadruplo que generará un salto a otro cuadruplo | (GOTO, None, None, 5). Este ejemplo, salta al cuádruplo 5. |
| GOTOF | Cuadruplo que generará un salto en el caso de que una condición sea en falso. | (GOTOF, t1, None, 8). Este ejemplo, salta al cuádruplo 8 en caso de que t1 resulte ser falso. |
| GOTOV | Cuádruplo que generara un salto en el caso de que una condición sea verdadero. | (GOTOV, t3, None, 9). Este ejemplo, salta al cuádruplo 9 en caso de que t3 resulte verdadero. |
| GOSUB | Código que hará el brinco a una función. | (GOSUB, None, None, 1). Este cuádruplo saltara al cuádruplo 1, en donde comienza una función. |
| ERA | Código de Expansión del Registro de Activación. | (ERA, iSuma, None, None). Este cuádruplo, prepara a memoria para el salto a la función iSuma. |
| PARAM | Código que parametriza para mandar a una función. | (PARAM, t1, None, p). En este ej mplo, t1, seria uno de los parámetros enviados a alguna función. |
| ENDPROC | Marca el fin de un procedimiento | (ENDPROC, None, None, None) |
| RETURN | Marca el valor de retorno de una función | (RETURN, t3, None, None) |

##### Diagramas de sintáxis

##### Descripción de las acciones semánticas

Las acciones semánticas, son aquellas en las que tomamos una acción importante dentro de la sintaxis, para la generación de cuadruplos.

* gotoMain: se crea y guarda el valor del numero de cuádruplo al inicio de la compilación, en el momento de aparecer el token “main”, se completa con la dirección a saltar.
* gotoSub: se crea al momento de realizarse una llamada a una función. Previamente, cuando se declarlo la función se guardo el numero de cuádruplo a la que saltara
* gotoF: se crea después de evaluar una expresión dentro de una condicional, y se le completa el valor de salto al momento en que aparece el token “else”.
* Goto: se crea en el momento en que se completa el gotoF, y se completa su dirección de salto cuando se termina el estatuto condicional
* GotoV: se utiliza en los ciclos do-while, se genera cuando apareceel token “do”, y se pone después de la condición del ciclo.

##### Tabla de consideraciones semánticas

Esta es la estructura del cubo semántico, en donde, en base a los dos tipos de datos de la operación en curso, regresa el valor de retorno, o en su caso, error.

cuboSemantico = {

"int": {

"int": {

"+":"int",

"-":"int",

"\*":"int",

"/":"int",

">":"bool",

">=":"bool",

"<=":"bool",

"<":"bool",

"=":"int",

"==": "bool",

"<>":"bool",

"and":"Error",

"or":"Error",

},

"float": {

"+":"float",

"-":"float",

"\*":"float",

"/":"float",

">":"bool",

">=":"bool",

"<=":"bool",

"<":"bool",

"=":"int",

"==": "bool",

"<>":"bool",

"and":"Error",

"or":"Error",

},

"bool": {

"+":"Error",

"-":"Error",

"\*":"Error",

"/":"Error",

">":"Error",

">=":"Error",

"<=":"Error",

"<":"Error",

"=":"Error",

"==": "Error",

"<>":"Error",

"and":"Error",

"or":"Error",

},

"string": {

"+":"Error",

"-":"Error",

"\*":"Error",

"/":"Error",

">":"Error",

">=":"Error",

"<=":"Error",

"<":"Error",

"=":"Error",

"==": "Error",

"<>":"Error",

"and":"Error",

"or":"Error",

},

},

"float":{

"int": {

"+":"float",

"-":"float",

"\*":"float",

"/":"float",

">":"bool",

">=":"bool",

"<=":"bool",

"<":"bool",

"=":"float",

"==": "bool",

"<>":"bool",

"and":"Error",

"or":"Error",

},

"float": {

"+":"float",

"-":"float",

"\*":"float",

"/":"float",

">":"bool",

">=":"bool",

"<=":"bool",

"<":"bool",

"=":"float",

"==": "bool",

"<>":"bool",

"and":"Error",

"or":"Error",

},

"bool": {

"+":"Error",

"-":"Error",

"\*":"Error",

"/":"Error",

">":"Error",

">=":"Error",

"<=":"Error",

"<":"Error",

"=":"Error",

"==":"Error",

"<>":"Error",

"and":"Error",

"or":"Error",

},

"string": {

"+":"Error",

"-":"Error",

"\*":"Error",

"/":"Error",

">":"Error",

">=":"Error",

"<=":"Error",

"<":"Error",

"=":"Error",

"==":"Error",

"<>":"Error",

"and":"Error",

"or":"Error",

},

},

"bool":{

"int": {

"+":"Error",

"-":"Error",

"\*":"Error",

"/":"Error",

">":"Error",

">=":"Error",

"<=":"Error",

"<":"Error",

"=":"Error",

"==":"Error",

"<>":"Error",

"and":"Error",

"or":"Error",

},

"float": {

"+":"Error",

"-":"Error",

"\*":"Error",

"/":"Error",

">":"Error",

">=":"Error",

"<=":"Error",

"<":"Error",

"=":"Error",

"==":"Error",

"<>":"Error",

"and":"Error",

"or":"Error",

},

"bool": {

"+":"Error",

"-":"Error",

"\*":"Error",

"/":"Error",

">":"Error",

">=":"Error",

"<=":"Error",

"<":"Error",

"=":"bool",

"==":"bool",

"<>":"bool",

"and":"bool",

"or":"bool",

},

"string": {

"+":"Error",

"-":"Error",

"\*":"Error",

"/":"Error",

">":"Error",

">=":"Error",

"<=":"Error",

"<":"Error",

"=":"Error",

"==":"Error",

"<>":"Error",

"and":"Error",

"or":"Error",

},

},

"string":{

"int": {

"+":"Error",

"-":"Error",

"\*":"Error",

"/":"Error",

">":"Error",

">=":"Error",

"<=":"Error",

"<":"Error",

"=":"Error",

"==":"Error",

"<>":"Error",

"and":"Error",

"or":"Error",

},

"float": {

"+":"Error",

"-":"Error",

"\*":"Error",

"/":"Error",

">":"Error",

">=":"Error",

"<=":"Error",

"<":"Error",

"=":"Error",

"==":"Error",

"<>":"Error",

"and":"Error",

"or":"Error",

},

"bool": {

"+":"Error",

"-":"Error",

"\*":"Error",

"/":"Error",

">":"Error",

">=":"Error",

"<=":"Error",

"<":"Error",

"=":"Error",

"==":"Error",

"<>":"Error",

"and":"Error",

"or":"Error",

},

"string": {

"+":"Error",

"-":"Error",

"\*":"Error",

"/":"Error",

">":"Error",

">=":"Error",

"<=":"Error",

"<":"Error",

"=":"string",

"==":"bool",

"<>":"bool",

"and":"Error",

"or":"Error",

},

},

}

#### Administración de memoria(descripción gráfica)

Para la administración de memoria, se utilizaron los diccionarios de Python, los cuales nos permiten ir indexando sub-diccionarios a la definición ya existente.

##### Directorio de Funciones

{'main':

{'Vars': {},

'Tipo': 'Main'

},

funcion:

{'Vars': {},

'Tipo': 'programa'

}

}

##### Tablas de Variables

Por ejemplo, en este diccionario, mostramos la tabla de variables global, la cual en estos instantes solo consta de una variable tipo entera, a partir de la dirección de memoria 1000 y que no cuenta con dimensión.

{programa:

{'Vars':

{iA:

{'Scope': 'global',

'Dir': 1000,

'TipoVar': u'int',

'Dim': None

}

},

'Tipo': 'programa'

}

* + - 1. Cuádruplos

Los cuádruplos, los fuimos almacenando mientras se generaban en una lista, indexándolos cada vez que se creaba un nuevo Cuádruplo. Después, en memoria virtual, se fueron recorriendo uno por uno para realizar las acciones necesarias.

|  |  |  |  |
| --- | --- | --- | --- |
| + | a | b | T1 |
| = | T1 |  | Resp |
| write | resp |  |  |

### Descripción de la máquina virtual

#### Descripción de Administración de Memoria en ejecución

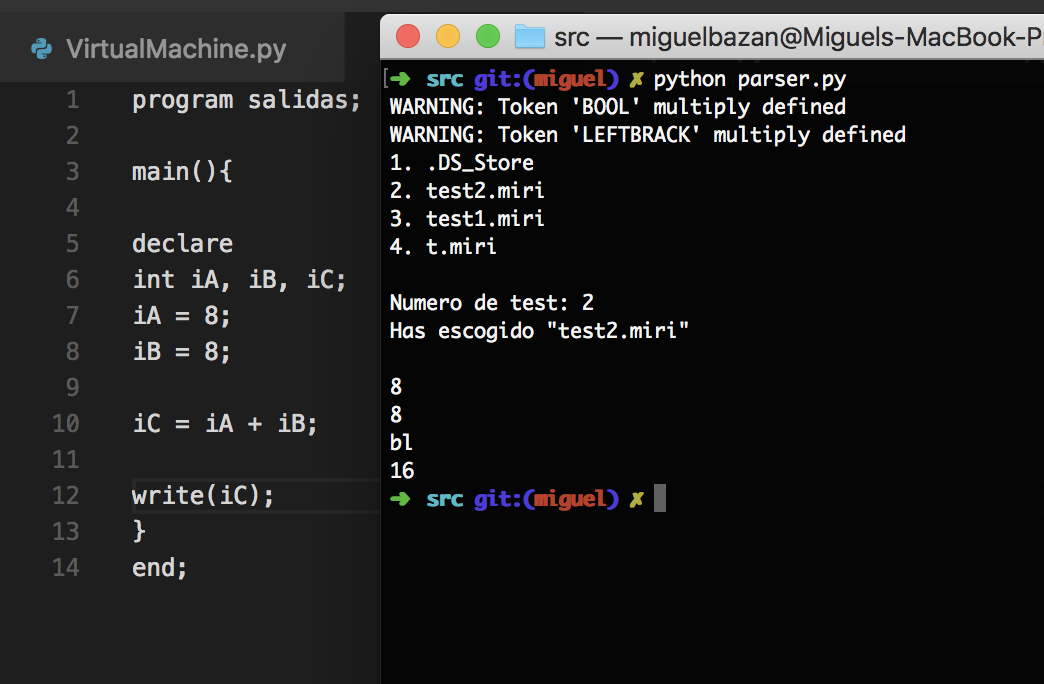
Inicios de memoria para cada tipo de dato basados en su scope (global, local, temporal)



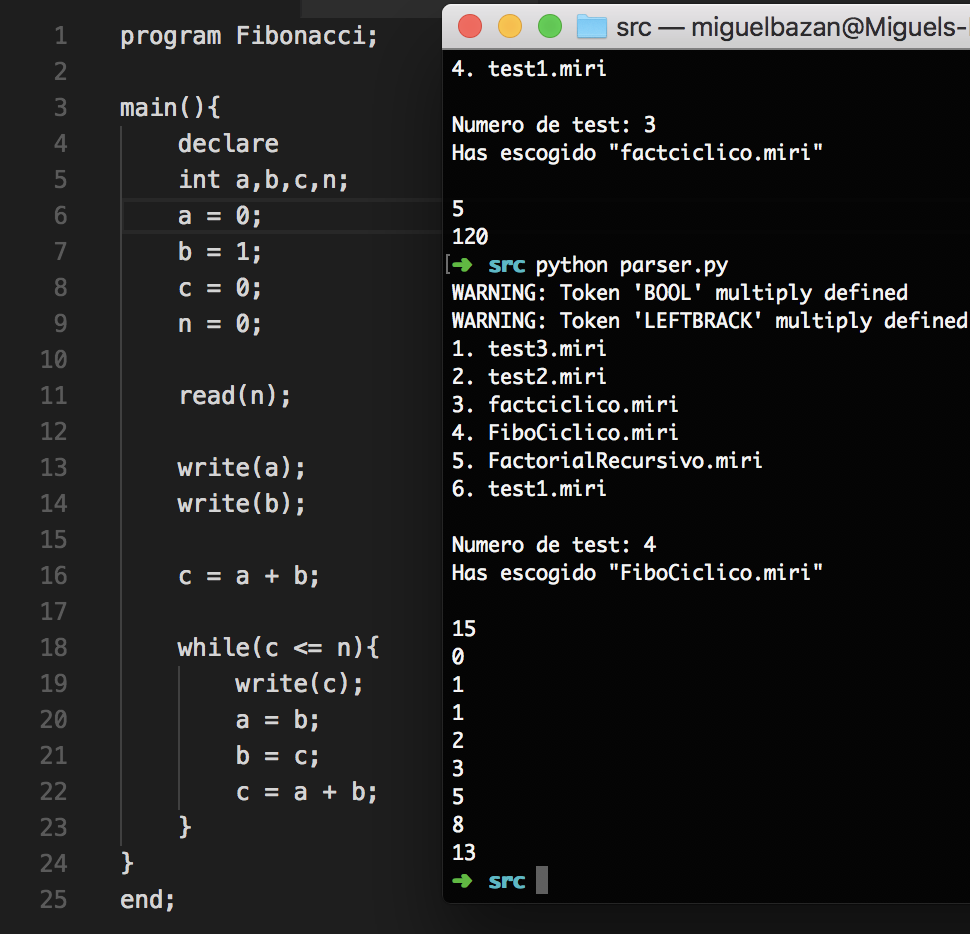
Se tienen vectores dinámicos para la creación de la memoria para cada scope (global, local y temporal) dependiendo de su tipo de dato (int, string, float,bool)

### Pruebas del funcionamiento del lenguaje

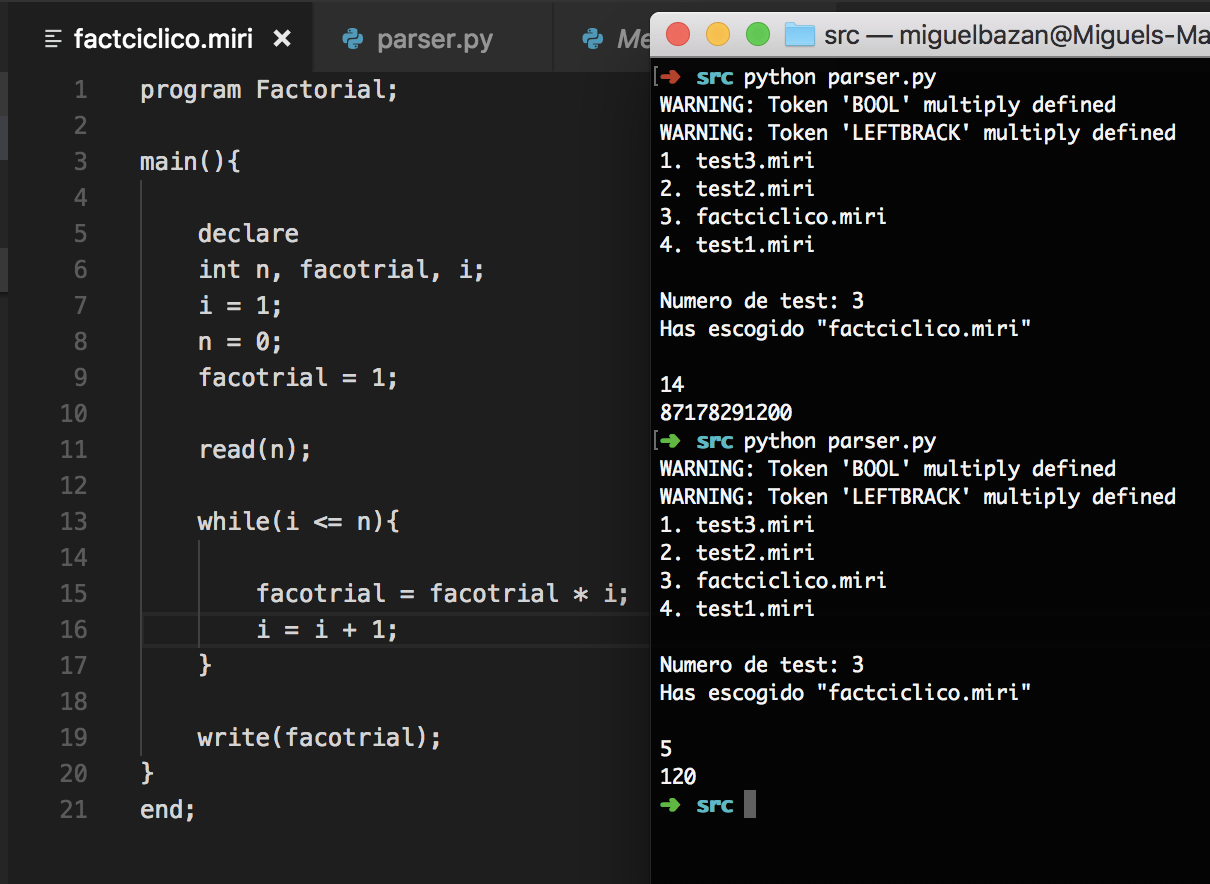
#### Codificación y resultados de la prueba



SUMA



Fibonacci ciclico



Factorial ciclico

### Listados del proyecto

#### Comentarios de documentación

############################################################

#Nombre del archivo: parser.py

#Autores:

# Ricardo Licea Mata A01280892

# Miguel Bazán Aviña A01281010

#

#Función del archivo:

#Este archivo corresponde al analizador sintáctico del compilador.

#A través de producciones, define la gramática del compilador, importando

#los tokens del analizador léxico. Aquí, están insertados las acciones

#semánticas para la generación de código intermedio, así como la creación

#e indexamiento de las tablas de variables.

#

#

############################################################

#Nombre del archivo: structs.py

#Autores:

# Ricardo Licea Mata A01280892

# Miguel Bazán Aviña A01281010

#

#Función del archivo:

#Este archivo corresponde a las estructuras de datos que utilizamos

#dentro del desarrollo del proyecto. Tiene dos clases: Queue y Stack,

#que son las estructuras usadas, por ejemplo, en las pilas de Operadores,

#Operandos y Saltos para generación de código intermedio

#

############################################################

#Nombre del archivo: turtle.py

#Autores:

# Ricardo Licea Mata A01280892

# Miguel Bazán Aviña A01281010

#

#Función del archivo:

#Este archivo corresponde a las llamadas a turtle, un modulo de python

#que permite mandar a llamar a las funciones, dándoles los parámetros necesarios,

#para generar una salida gráfica. Este archivo, es utilizado por la máquina virtual para

#generar las salidas gráficas.

#

#

############################################################

#Nombre del archivo: VirtualMachine.py

#Autores:

# Ricardo Licea Mata A01280892

# Miguel Bazán Aviña A01281010

#

#Función del archivo:

#

# La función principal de la maquina virtual es manejar los cuádruplos como

# entrada y de aquí partir a generar las operaciones que se necesiten para ejecutar

# todo el código intermedio. Se lee de memoria y se asigna a memoria.

#

# Dependiendo de los parámetros que se reciben son las acciones

# que se toman para cada uno de los cuádruplos.

############################################################

#Nombre del archivo: MemoriaV.py

#Autores:

# Ricardo Licea Mata A01280892

# Miguel Bazán Aviña A01281010

#

#Función del archivo:

#Este archivo corresponde a la asignación de memoria en el compilador.

#Cuando recibe una variable, arreglo o temporal, dependiendo su tipo,

#les asigna una dirección numérica que posteriormente es guardada en la tabla de variables

#de cada función, asociándola con la variable en cuestión.

#

#

############################################################

#Nombre del archivo: cuboSemantico.py

#Autores:

# Ricardo Licea Mata A01280892

# Miguel Bazán Aviña A01281010

#

#Función del archivo:

#Este archivo, recibe en sus funciones 3 variables, los

# cuales corresponderán a los tipos de datos involucrados en

#alguna operación y al operador en si. Dependiendo los tipos de datos,

# y el operador, regresa el valor de retorno que debería de tener dicha operación

#o en su caso, una Key de error especificando que la operación entre esos tipos de datos

#no es posible.

#

## Manual de usuario

Para comenzar a programar en el lenguaje miri se necesita definir un nombre de programa y que cuente con una función main. La manera de hacer esto es:

*program prueba;*

*main(){*

*}*

*end;*

Es necesario definir la palabra reservada “program” antes del nombre del programa y después poner un “;”. Seguido de esta definición, se necesita tener un main para correr las funciones.

El lenguaje permite definir variables de tipo int, float, bool y string. Para definir variables se debe escribir la palabra “declare”. Y debajo de esta poner las definiciones de variables.

Nótese que en la palabra declare no se necesita un “;”.

*program prueba;*

*main(){*

*declare*

*int iA = 8;*

*float iF = 8.4;*

*bool iB = true;*

*string iS = “Hola”;*

*}*

*end;*

El “;” tiene que ir al final de cada línea excepto en la declaración de funciones.

Para definir una función se necesita crear antes de la función main. Si se hace después no se aceptara. La función se define poniendo el tipo de dato y luego un nombre de función.

*program prueba;*

*int iSuma(int iA){*

*declare*

*int iJ, iRes;*

*iRes = iA + iJ;*

*return iRes;*

*}*

*main(){*

*declare*

*int iRespuesta,iB;*

*iB = 89;*

*iRespuesta = iSuma(iB);*

*}*

*end;*

El lenguaje también permite el uso de condicional IF. El IF tiene que ir siempre seguido de un ELSE.

*program prueba;*

*int iSuma(int iA){*

*declare*

*int iJ, iRes;*

*iRes = iA + iJ;*

*return iRes;*

*}*

*main(){*

*declare*

*int iRespuesta,iB, iA, iC;*

*iB = 89;*

*if (iA < iC){*

*iRespuesta = iSuma(iA, iC);*

*}else{*

*iRespuesta = iSuma(iB,iA);*

*}*

*}*

*end;*

Miri tambien permite el uso de ciclos. Para crear un ciclo se hace lo siguiente:

*program prueba;*

*main(){*

*declare*

*int iA, iB, iC,*

*iA = 8;*

*iB = 78;*

*iC = 0;*

*while(iA < iB)*

*}*

*end;*

MIRI, permite el uso de variables dimensionadas, es decir arreglos de 1 y 2 dimensiones:

*program prueba;*

*main(){*

*declare*

*inti A[5], iB[5][3], iC;*

*iC = A[2];*

*iC = B[3][2];*

*}*

*end;*

En este ejemplo, la variable iC es primero igualada al valor de la casilla 2 del arreglo A. Posteriormente, al valor que esta en la casilla del renglón 3, columna 2 de la matriz B.

La mayor atracción de MIRI, es su facilidad de generar salidas gráficas a través de los comandos *circulo, cuadro, triangulo y línea*.

*program prueba;*

*main(){*

*circulo(10, red, true, 8, 9);*

*cuadro(10, blue, true, 5, 6);*

*}*

*end;*

En este ejemplo, podemos ver como se les manda a llamar a las funciones definidas circulo y cuadro que pintan, dependiendo de sus parámetros, la figura gráficamente.