

Tecnológico de Monterrey - Campus Monterrey
Periodo: Febrero - Junio 2024



TC3003B.502
Desarrollo de aplicaciones avanzadas de ciencias computacionales
Grupo 501

Little-Duck

Roberto Calleja Pedraza
A01024834

Profesores
Elda Quiroga

Fecha:
29 de Abril del 2024

Scanner y Parser

Para el Scanner y el Parser se selecciono la herramienta Antlr, en conjunto con Python. La gramática se transcribió de la entrega pasada a un archivo llamado “little_duck.g4”. Al ejecutar este código, junto con la herramienta Antlr se generaron 7 archivos que en conjunto actúan como Scanner y Parser y estos archivos son llamados por el programa “Driver.py” para compilar los diferentes códigos.

La entrega cuenta con todos los códigos previamente mencionados, junto con una carpeta de test la cual cuenta con 7 archivos de tipo “.txt” los cuales contienen las diferentes pruebas que se realizaron. La información de como ejecutar los archivos se encuentra en el archivo README.txt.

Tabla de consideraciones semánticas (Cubo semántico)

Operando 1	Operando 2	Operadpr							
		+	-	*	/	>	<	!=	=
int	int	int	int	int	float	bool	bool	bool	int
int	float	float	float	float	float	bool	bool	bool	error
int	string	error	error	error	float	error	error	error	error
float	int	float	float	float	float	bool	bool	bool	error
float	float	float	float	float	float	bool	bool	bool	float
float	string	error	error	error	float	error	error	error	error
string	int	error	error	error	float	error	error	error	error
string	float	error	error	error	float	error	error	error	error
string	string	error	error	error	float	error	error	error	String

Para el lenguaje se implementó este cubo semántico, el cual cuenta con las relaciones entre los tipos de variable, int, float y string. Con esta tabla se implementó una función “check_cubo_semantico” la cual recibe los dos operandos y el operador, y da como resultado el tipo de valor que se espera de esa operación o en caso de operadores no compatibles se levanta una excepción.

Estructuras de datos

Directorio de Funciones y a las Tablas de Variables

Variable
+ name: str +type: str +value: str +dir: int

FunctionDirectory
+ functions: {Function} + global_var_table: VarTable
-add_function(name):void - add_variable(variable, function_name):void

Function
+ name: str + var_table: VarTable

VarTable
+variables: {Variable}

Durante la ejecución del Listener, se crea el directorio de funciones y se le van agregando las funciones apropiadas junto cos sus variable así como las variables globales. Las variables que se agregan tienen nombre y tipo, con valor y dirección nula.

Tabla de constantes

constant
+ value: str + type: str +dir: int

ConstantTable
+ constants: {}
-add_constant(value, type)

Así como el directorio de funciones estas objetos se crean y llenan durante la ejecución del Listener, creando una sola tabla de constantes y agregando el valor, y tipo y dejando en blanco la dirección.

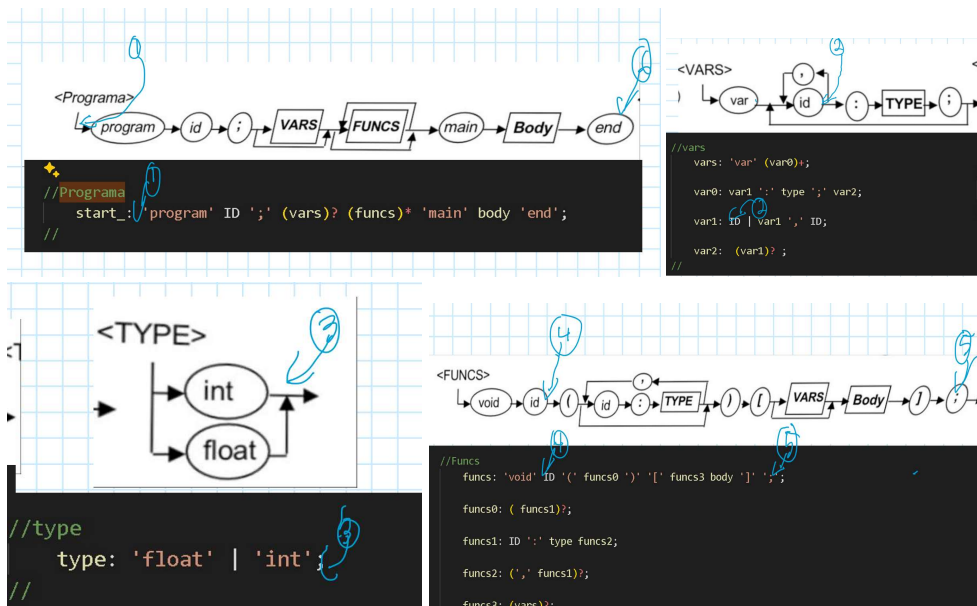
Cuadruplo

cuadruplo
+ operator: str + left: int #dir +right: int #dir +result: int #dir +scope

El manejo de los cuádruplos se realiza dentro del visistor (los puntos neurálgicos se encuentran abajo), se crean con las direcciones de las variables para representar las direcciones de los operandos (left, right y result) y se almacenan en un arreglo.

Puntos neurálgicos

Llenado de tabla de variables y constantes



1. Crear todas las variables y arreglos.

2. Agregar variables a current_var_arr

3. Leer Type y guardarlo en current_typr

3.1 Agregar variables a función_directory

3.2 Limpiar arreglo current_var_arr

4. Crea Función

4.1 Actualiza current_function

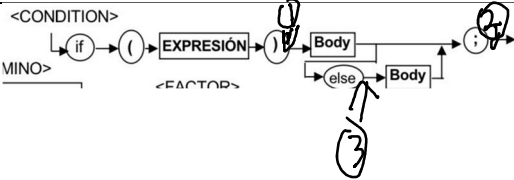
5. Actualiza current_function a global

6. Elimina todas las tablas e imprime la tabla

Estos puntos neurálgicos fueron implementados en el archivo little_duckListener.py, en las funciones de relacionadas a la gramática correspondiente, y cada punto esta marcado con las iniciales DFTV2-PN (Directorio de funciones y tablas de variables entrega 2 Puntos Neuralgicos) acompañado del numero de punto correspondiente

Creación de cuádruplos

	<ol style="list-style-type: none"> 1. Agrega "id" a pila de operandos 2. Agrega el operador a la pila de operadores 3. CreaCuádruplo()
	<ol style="list-style-type: none"> 1. Agrega el operador a la pila de operadores 2. CreaCuádruplo()
	<ol style="list-style-type: none"> 1. Agrega el operador a la pila de operadores 2. CreaCuádruplo()
	<ol style="list-style-type: none"> 1. Agrega el operador a la pila de operadores 2. CreaCuádruplo()
	<ol style="list-style-type: none"> 1. Agrega el operador a la pila de operadores Agrega 0 a la pila de operandos 2. Agrega id a la pila de operandos Crea el cuádruplo 3. Agrega "(" a la pila de operadores 4. Elimina "(" de la pila de operadores
	<ol style="list-style-type: none"> 1. Agrega el valor correspondiente a la pila de operandos
	<ol style="list-style-type: none"> 1. Agrega el string a la pila de operandos 2. Agrega print a pila de operadores 3. CreaCuádruplo() hasta que el operador no sea print Agrega un cuádruplo mas para indicar salto de línea
	<ol style="list-style-type: none"> 1. Agrega el contador de cuádruplos actual a la pila de saltos 2. Crea un cuádruplo GOTOV y como resultado agrega el ultimo valor de la pila de saltos

	<ol style="list-style-type: none"> 1. Agrega GOTOF a la pila de operadores y crea un cuádruplo Agrega el contador actual a la pila de saltos 2. Agrega la posición actual al cuádruplo en la posición del valor del ultimo elemento de la pila de saltos 3. Agrega GOTO a la pila de operadores y crea un cuádruplo Agrega la posición actual al cuádruplo en la posición del valor del ultimo elemento de la pila de saltos <p>Agrega el contador actual a la pila de saltos</p>
---	--

Manejo de memoria y maquina virtual

Manejo de memoria

Para el manejo de memoria, se crea un archivo llamado MemoryManager.py, que se ejecuta después de generar las tablas de variables en el listener. Este archivo recibe el directorio de funciones y la tabla de constantes, creando un arreglo compuesto por cuatro subarreglos, uno para cada tipo de dato: int, float, bool y string. Luego, recorre el directorio de funciones y la tabla de constantes, asignando los valores de estos datos a la memoria y llenando las clases de las variables con la dirección del dato, facilitando así la creación de cuádruplos.

Las variables temporales se agregan después de la creación de los cuádruplos, utilizando los contadores de variables temporales y agregando la cantidad exacta de espacios de memoria al arreglo.

VM

La máquina virtual consta de un programa tipo switch que recibe el arreglo de memoria y la lista de cuádruplos. La lista se recorre dentro de un ciclo while con un contador que avanza de uno en uno, excepto en los cuádruplos de saltos (GOTO, GOTOF, GOTOV), donde se realiza el salto al cuádruplo correspondiente.

Pruebas

Se generaron las siguiente 7 pruebas, en una carpeta llamada test:

- test1: test de programa
- test2: test de variable y asignación
- test3: test de print
- test4: test de Func #no se implemento
- test5: test de condicional
- test6: test de ciclo
- testF: test de integración

Estos archivos se ejecutan:

C:/.../ code> python Driver/Driver.py "nombre del archivo"

Por ejemplo: python Driver/Driver.py test/test1.txt

Gramática

Expresiones Regulares

INT: [0-9]+

FLOAT: [0-9]+ '.' [0-9]+

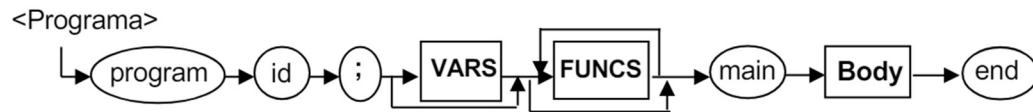
STRING: "[^"]*"

ID: [a-zA-Z0-9]*

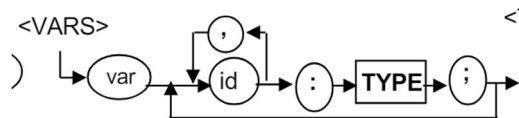
Lista de tokens

Main	Program	While
End	Id	Do
Var	Cte.String	If
Print	Cte.Int	Else
void	Cte.float	int
float	[]
()	{
}	;	:
,	+	-
*	/	=
<	>	!=

Gramática libre de contexto



<programa> -> program id ; <vars> <funcs> main <bodyend>



<vars> -> var < var0>

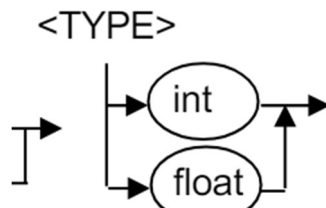
<var0> -> <var1>: <type>; <var2>

<var1> -> id

<var1> -> id, <var1>

<var2> -> E

<var2> <var1>



<type> - float

<type> -> int



<body> -> { <body0>

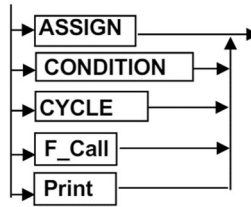
<body0> -> }

<body0> -> <statement> < body1> }

<body1> -> < statement> <body1>

<body1> -> E

<STATEMENT>



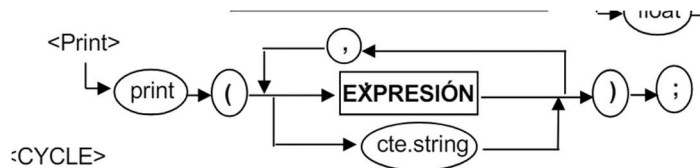
<statement> -> <assign>

<statement> -> <condition>

<statement> -> < cycle>

<statement> -> <F_Call>

<statement> -> <print>



<CYCLE>

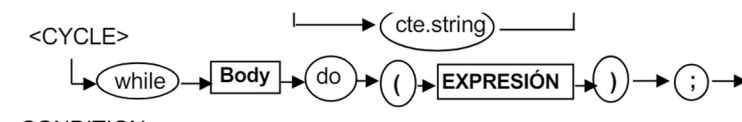
<print> -> print (print0);

<print0> -> <expresion> <print1>

<print0> -> cte.string <print1>

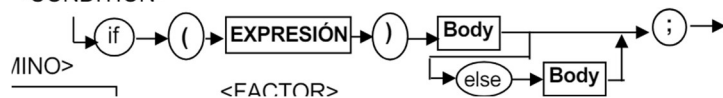
<print1> -> E

<print1> -> , <print0>



<cycle> -> while <body> do (<expression>);

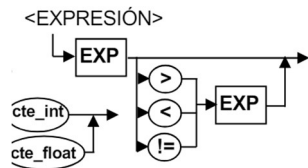
<CONDITION>



<condition> -> if (<expresión>) <body> <condition0>

<condition0> -> E

< condition0> -> else <body>



<expresión> -> <exp> < expresión0>

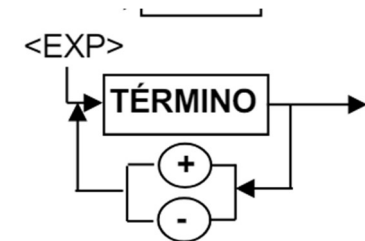
< expresión0> -> E

< expresión0> -> < expresión1> <exp>

< expresión1> -> >

< expresión1>-> <

< expresión1> -> !=



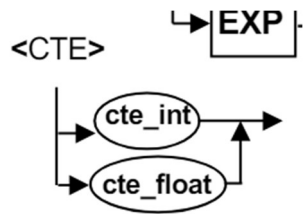
< exp> -> <termino> <exp0>

<exp0> -> E

<exp0> -> <exp1> <exp>

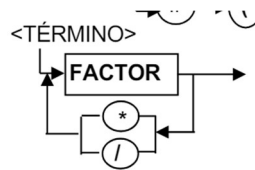
<exp1> -> -

<exp1> -> +



<cte> -> cte_float

<cte> -> cte_int



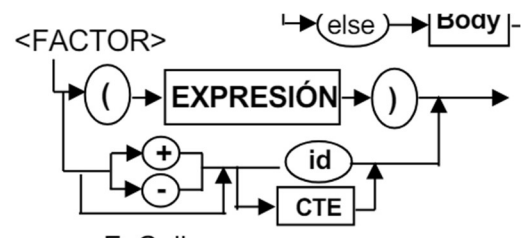
<termino> -> < factor> < termino0>

< termino0> -> E

< termino0> -> < termino1> < termino>

< termino1>-> *

< termino1>-> /



<factor> -> (< expresión>)

<factor> -> <factor0> <factor1>

<factor0> -> +

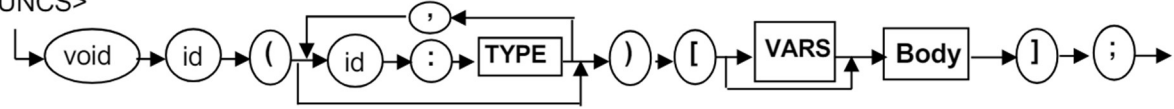
<factor0>-> -

<factor0>-> E

<factor1>-> id

<factor1>-> <cte>

<FUNCS>



<Funcs> -> void id (<Funcs0>) [<Func3> < body>];

<Funcs0> -> E

<Funcs0>-> <Funcs1>

<Funcs1>-> id :<type> < Funcs2>

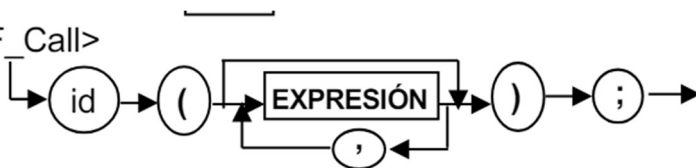
<Funcs2>-> E

<Funcs2>-> , <Funcs1>

<Funcs3> -> <vars>

<Funcs3> -> E

<F_Call>



<f_call> -> id (< f_call0>);

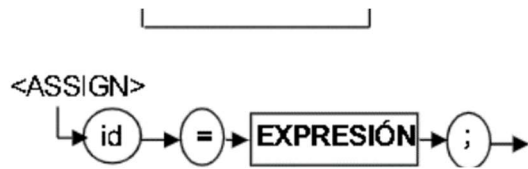
< f_call0> -> E

< f_call0> -> < f_call1>

<f_call1> -> <expression> < f_call 2>

< f_call2> -> E

<f_call2> -> , < f_call2>



<assign> -> id = <Expresion> ;