Diseño de compiladores

Maestra Elda Quiroga

Proyecto: Shinto

Sergio González Sifuentes

A00821229

06/06/2022

INDICE

Descripción del Proyecto:

El propósito de este lenguaje es evaluar los conocimientos del alumno en base a lo aprendido en la clase, el objetivo fue desarrollar un lenguaje básico con estatutos, variables, funciones, lectura, escritura y expresiones. Usando todo lo aprendido en clase: léxico, semántica, sintáxis, creación de código y ejecución

Caso de uso: UU001	**Actor primario:** Usuario
Descripción: Se asignará un dato a una variable	
Condición de éxito La variable es asignada con éxito	
Workflow: 1. Se ejecuta el compilador 2. Se definirá dentro de una función o globalmente una variable con tipo y nombre de la sig manera: var (tipo de dato) (nombre de la variable) ; (Pueden declararse n variables del mismo tipo con diferentes nombres en una misma linea 3. El compilador crea una variable con esos datos, creandole una dirección y almacenándola en el directorio de variables como una instancia de la clase Variable	
Precondición Ninguna	
Restricciones y especificación No debe existir una variable con ese nombre	

Caso de uso: UU002	**Actor primario:** Usuario

Descripción: Se realizarán operaciones aritmeticas
Condición de éxito La respuesta mostrara un resultado siguiendo la jerarquía de operaciones
Workflow: 1. Se ejecuta el compilador 2. Se escribe una expresión con sintáxis correcta 3. El compilador va guardando las operaciones en el stack y los operadores en otro para poder formar los cuadruplos correspondientes
Precondición Ninguna
Restricciones y especificación Ninguna

Caso de uso: UU003	**Actor primario:** Usuario
Descripción: Se declarará una función	
Condición de éxito El compilador ejecuta con éxito la función	

Workflow:

1. Se crea una función de la siguiente manera: (tipo de dato) function (nombre de la función) ((parámetros si es que los hay escritos como (nombre del parámetro, tipo de dato))) { (estatutos si es que los hay) }
2. Se crea la función y se agrega al directorio de funciones con una dirección
3. Se agregan los parámetros al directorio de funciones y se le agregan al diccionario de parámetros dentro de su función correspondiente
4. Se agregan las variables que se declaran en el inicio de la función al directorio de variables y se le agregan al diccionario de parámetros
5. Se crean los cuadruplos de los statements dentro de la función
6. Cerramos la función y si no hubo ningún problema, seguimos

Precondición
Ninguna

Restricciones y especificación
Ninguna

Caso de uso: UU004	**Actor primario:** Usuario

Descripción:
Se regresara un valor en una función

Condición de éxito
Se regresa con éxito el valor esperado

Workflow:

1. Se ejecuta el compilador
2. Se escribe una función
3. Se guarda en una variable el valor de la función al llamarla
4. Se regresa el valor de retorno de una función

Precondición Ninguna
Restricciones y especificación Ninguna

Caso de uso: UU005	**Actor primario:** Usuario
Descripción: Se leerá un número y se guardará en una variable	
Condición de éxito Al teclear la variable se regresará lo que se había tecleado anteriormente	
Workflow: 1. Se ejecuta el compilador 2. Se declara input(ID) (ID siendo el nombre de una variable 3. El programa pedirá input 4. Se teclea un número 5. El programa guardará el valor que se tecleó en la variable	
Precondición Ninguna	
Restricciones y especificación Ninguna	

Primero se desarrollo junto con sly la generación de tokens, la detección de palabras, números, etc. Además de retornar los valores de los int, float y booleans para después poder trabajar con ellos más adelante.

Se definieron los siguientes tokens:

```
tokens = {
    'PROG',
    'MAIN',
    'VAR',
    'FUNC',              # func
    'ID',
    'INT',
    'FLOAT',
    'STRING',
    'BOOL',
    'TRUE',
    'FALSE',
    'D_INT',
    'D_FLOAT',
    'D_STRING',
    'D_BOOL',
    'IF',                # if
    'THEN',              # then
    'ELSE',              # else
    'FOR',               # for
    'TO',                # to
    'WHILE',             # while
    'ARROW',             # arrow
    'COMMENT',           # //
    'EQEQ',              # ==
    'GOETHAN',          # >=
    'LOETHAN',          # <=
    'DIFF',             # !=
    'AND',              # &&
    'OR',                # ||
    'RETURN'
}
```

Además, se definió un diccionario para reservar palabras del lenguaje, palabras clave que se muestran a continuación:

```
reserved = {
    'program'       : 'PROGRAM',
    'main'          : 'MAIN',
    'var'           : 'VAR',
    'if'            : 'IF',
    'else'          : 'ELSE',
    'function'      : 'FUNC',
    'return'        : 'RETURN',
    'input'         : 'INPUT',
    'print'        : 'OUTPUT',
    'int'           : 'INT',
    'float'         : 'FLOAT',
    'string'        : 'STRING',
    'bool'          : 'BOOL',
    'while'         : 'WHILE'
}
```

También, sly ayudó a poder definir literales y que no se tuviesen que generar tokens para ellos, los cuales se muestran a continuación:

```
literals = {
    '=',
    '+',
    '-',
    '*',
    '/',
    '(',
    ')',
    ',',
    ';',
    '.',
    ' ',
    ':',
    '[',
    ']',
    '{',
    '}'
   }
```

También se definieron las reglas para generar los distintos tokens y que se identificaran correctamente:

```
@_(r'bool')
def BOOL(self, t):
    t.value = str(t.value)
    return t

@_(r'true')
def TRUE(self, t):
    t.value = str(t.value)
    return t

@_(r'false')
def FALSE(self, t):
    t.value = str(t.value)
    return t

@_(r'\".*?\"')
def STRING(self, t):
    t.value = str(t.value)
    return t

@_(r'\d+\.\d*')
def FLOAT(self, t):
    t.value = float(t.value)
    return t

@_(r'\d+')
def INT(self, t):
    t.value = int(t.value)
    return t

@_(r'[a-zA-Z_][a-zA-Z_0-9]*')
def ID(self, t):
    t.type = self.reserved.get(t.value, 'ID')
    return t
```

Primero se hizo una clase de Tipos de Dato que me ayudaría a trabajar fácilmente con la manipulación de datos, después con eso pude empezar a trabajar con el Cubo Semántico en el que podríamos hacer match con cada tipo de dato y el match regresaría un tipo de dato y si no fuera posible, se regresaría un tipo de dato inválido.

Antes de comenzar a trabajar de lleno en la gramática decidí empezar a trabajar en como se almacenarían los datos que se van tomando y generando.

Se hicieron 3 clases de tablas global, local y de constantes, como sus nombres lo dicen global sería para los datos que fueron inicializados en el scope global, local serían los

que fueron definidos dentro de una función y la tabla de constantes sería para las constantes que vayamos usando y que necesitemos guardar.

Antes de poder almacenar en ellas, se tenían que delimitar la cantidad de almacenamiento que tendría cada diccionario de cada tabla así que se hizo una clase Delimitation que almacenaría el lugar en el que empieza cada tipo de dato de cada tabla y el numero de entradas de ese tipo que vamos llenando.

Para cada tabla se hicieron 4 diccionarios para int, float, string y booleans en el que cada uno haría la verificación de que no sea repetido antes de insertarlo en el diccionario.

Después se hizo una clase llamada Variables con los siguientes atributos:

```
class Variable:
    name: str
    data_type: str
    value: str = "Null"
    addr: int
    dimensions: int
    spaces: int
    scope: str
```

Y también una clase llamada Función con los siguientes atributos:

```
class Function:
    name: str
    data_type: str
    returnVar: bool
    initQuad: int
    addr: int
```

La cual también incluye un diccionario de parámetros y un diccionario de variables

Después se hizo una clase para la Jerarquía de operadores para poder hacer las operaciones de manera ordenada y que resulten bien a la hora de meterlas al stack de operadores y para poder identificarlas más fácilmente.

Ahora, se empezó a trabajar en la clase de Memoria de Ejecución donde se escribiría donde va cada variable dependiendo de su scope (global, constante o local) y su tipo de dato (int, float, string y bool), además un método para convertir el dato almacenado en su tipo de dato correspondiente, se consigue los números de espacios que se necesitan en cada tipo de scope que vamos a conseguir sacando la longitud del diccionario que crearemos dentro del Parser que aun no codificamos pero para irnos

haciendo la idea de como trabajaremos, un método para sacar un valor de una dirección en particular y por último un método para guardar un valor en una dirección.

Después, trabajaremos en el VM, donde tendremos una instancia de la memoria de ejecución, una instancia de la jerarquía de operadores un arreglo vacio de los cuadruplos y una variable ip (instruction pointer) para poder movernos atraves en la ejecución, al iniciar se recibirá un diccionario que el parser generará donde recibiremos un diccionario de globales, constantes, locales y los cuadruplos, después se ejecutara cada uno de ellos. En el método de ejecución y resolución de cuadruplos se ira identificando el tipo de operación con ayuda de la clase de Jerarquía para identificarlos, reconocer la operación, realizarla, guardar el dato correspondiente en la dirección solicitada y por último aumentar la dirección del ip para seguir o en caso de tener un goto o un gotof ir al num de cuádruplo solicitado.

Ahora para el parser que vendría siendo lo más importante a mi parecer se inicializa con las siguientes variables:

```
class ShintoParser(Parser):
    debugfile = 'parser.out'
    tokens = ShintoLexer.tokens
    ERROR_FLAG = False
    stack_dim = []
    stack_params = []
    stack_vars = []
    stack_gvars = []

    call_params = []
    counter_params = 0
    counter = 0
```

Un debugfile que pertenece al sly y le damos el nombre de un archivo para que lo cree o lo reescriba si es que ya existe donde pondrá el camino que se utilizó en la gramática así como las reglas, meramente para hacer debugging, un arreglo de tokens que son los que definimos en el leer, un ERROR_FLAG para identificar errores, un stack de dimensiones para manejar los arrays, un stack de parámetros para cuando trabajemos con las funciones y los vayamos a adjuntar a las funciones, un stack de variables para declarar variables y un stack de gvars para declarar variables globales, un arreglo de call_params para poder trabajar con llamadas de funciones, un contador de parámetros igualmente para comparar el num de parámetros llamados con los declarados y un contador en general.

```
globals = G_Table()
locals = L_Table()
constants = C_Table()
quads = QuadOverseer()
delimitation = Delimitation()
dir_functions = Directory_Func()
dir_vars = Directory_Vars()
```

También tenemos todas estas instancias de lo que se trabajó previamente para poder trabajar con ellos.

Para cada gramática se definirán los tokens que se deben utilizar y funciones que necesitan vacíos para poder hacer cálculos y generar los cuadruplos.

Al final se parseara la data para hacer un diccionario que pueda ser utilizado por la VM.

```
def parseData(self) -> dict:
    data = {
        "Globals": self.globals.getGTable(),
        "Constants": self.constants.getCTable(),
        "Locals": self.locals.getLTable(),
        "Quadruples": self.quads.getQuads()
    }
    return data
```

Errores:

Los errores que se pueden encontrar son:

Al hacer un cuádruplo se tenga un error de que el polish vector esté vacío

Descripción del Compilador:

Se utilizó la pc personal para trabajar en este compilador, usando Visual Studio Code, con el lenguaje Python y varias extensiones de VSCode como Python, pylint, github. Se utilizó Sly que es una implementación de las herramientas lex y yacc con documentación decente y métodos muy útiles para desarrollar el Parser y el Lexer del lenguaje usando un algoritmo LALR(1)

Análisis del Léxico:

```
# Define keywords
PROG = r'program'
MAIN = r'main'
VAR = r'var'
FUNC = r'function'
IF = r'if'
THEN = r'then'
ELSE = r'else'
FOR = r'for'
TO = r'to'
WHILE = r'while'
ARROW = r'->'
OUTPUT = r'output'

EQEQ = r'=='
GOETHAN = r'>='
LOETHAN = r'<='
DIFF = r'!='
AND = r'&&'
OR = r'\|\|'
```

```
@_(r'int')
def D_INT(self, t):
    t.value = str(t.value)
    return t

@_(r'float')
def D_FLOAT(self, t):
    t.value = str(t.value)
    return t

@_(r'string')
def D_STRING(self, t):
    t.value = str(t.value)
    return t

@_(r'bool')
def BOOL(self, t):
    t.value = str(t.value)
    return t

@_(r'true')
def TRUE(self, t):
    t.value = str(t.value)
    return t

@_(r'false')
def FALSE(self, t):
    t.value = str(t.value)
    return t

@_(r'\".*?\"')
def STRING(self, t):
    t.value = str(t.value)
    return t

@_(r'\d+\.\d*')
def FLOAT(self, t):
    t.value = float(t.value)
    return t
```

```
@_(r'\d+')
def INT(self, t):
    t.value = int(t.value)
    return t

@_(r'[a-zA-Z_][a-zA-Z_0-9]*')
def ID(self, t):
    t.type = self.reserved.get(t.value, 'ID')
    return t

@_(r'//.*')
def COMMENT(self, t):
    pass

@_(r' ')
def space(self, t):
    pass

@_(r'\n+')
def ignore_newline(self, t):
    self.lineno += len(t.value)

# Compute column
def find_column(text, token):
    last_cr = text.rfind('\n', 0, token.index)
    if last_cr < 0:
        last_cr = 0
    column = (token.index - last_cr) + 1
    return column

# Error handling rule
def error(self, t):
    print("Illegal character '%s'" % t.value[0])
    self.index += 1
```

Análisis de Sintáxis:

Grammar:

Rule 0 S' -> program

Rule 1 program -> PROG ID check_program ; gvars store_gvars functions gvars store_gvars main

Rule 2 gvars -> VAR datatype gvarids store_gtype ; gvars

Rule 3 gvars -> <empty>

Rule 4 gvarids -> ID

Rule 5 gvarids -> ID , gvarids

Rule 6 store_gvars -> <empty>

Rule 7 store_gtype -> <empty>

Rule 8 vars -> VAR datatype varids store_type ; vars

Rule 9 vars -> <empty>

Rule 10 varids -> ID darray

Rule 11 varids -> ID darray , varids

Rule 12 store_type -> <empty>

Rule 13 darray -> <empty>

Rule 14 darray -> [INT] twodarray

Rule 15 twodarray -> <empty>

Rule 16 twodarray -> [INT]

Rule 17 functions -> datatype FUNC ID (params) store_funcv store_params store_init_quad { vars store_local_vars funcontent } close_func functions

Rule 18 functions -> <empty>

Rule 19 funcontent -> <empty>

Rule 20 funcontent -> statement funcontent

Rule 21 params -> ID : datatype , params

Rule 22 params -> ID : datatype

Rule 23 params -> <empty>

Rule 24 main -> FUNC MAIN ( ) store_funcm { vars store_mainv maincontent }

Rule 25 maincontent -> <empty>

Rule 26 maincontent -> statement maincontent

Rule 27 statement -> loop unload_pv

Rule 28 statement -> input unload_pv

Rule 29 statement -> output unload_pv

Rule 30 statement -> ifelse unload_pv

Rule 31 statement -> returns unload_pv ;

Rule 32 statement -> expr ;

Rule 33 statement -> var_assign unload_pv

Rule 34 statement -> <empty>

Rule 35 statement -> statement statement

Rule 36 var_assign -> ID store_oper = expr ;

Rule 37 returns -> RETURN expr store_rquad

Rule 38 ifelse -> IF (expr) store_gotof { statement } ELSE store_endif { statement }

Rule 39 ifelse -> IF (expr) store_gotof { statement } store_endif

Rule 40 output -> OUTPUT (expr outex) ;

Rule 41 outex -> <empty>

Rule 42 outex -> , expr outex

Rule 43 input -> INPUT (ID store_oper) ;

Rule 44 loop -> WHILE store_jump (expr) store_gotof { statement } end_loop

Rule 45 expr -> arexp exprx

Rule 46 exprx -> <empty>

Rule 47 exprx -> exprop arexp

Rule 48 exprop -> OR

Rule 49 exprop -> EQEQ

Rule 50 exprop -> DIFF

Rule 51 exprop -> AND

Rule 52 exprop -> <

Rule 53 exprop -> LOETHAN

Rule 54 exprop -> >

Rule 55 exprop -> GOETHAN

Rule 56 arexp -> term arexpextra

Rule 57 arexp -> term

Rule 58 arexpextra -> <empty>

Rule 59 arexpextra -> - term arexpextra [precedence=left, level=1]

Rule 60 arexpextra -> + term arexpextra [precedence=left, level=1]

Rule 61 term -> factor termx

Rule 62 term -> factor

Rule 63 termx -> <empty>

Rule 64 termx -> / factor termx [precedence=left, level=2]

Rule 65 termx -> * factor termx [precedence=left, level=2]

Rule 66 factor -> element

Rule 67 factor -> (store_op expr) store_op

Rule 68 element -> callfunc store_oper

Rule 69 element -> compound store_oper

Rule 70 element -> const store_const

Rule 71 compound -> compoundx

Rule 72 compoundx -> ID store_oper

Rule 73 const -> STRING

Rule 74 const -> FLOAT

Rule 75 const -> INT

Rule 76 const -> FALSE

Rule 77 const -> TRUE

Rule 78 callfunc -> ID verify_func add_fstack (callfuncpar ver_params) end_fstack store_gosub

Rule 79 callfuncpar -> <empty>

Rule 80 callfuncpar -> expr store_pquad callfuncparx

Rule 81 callfuncparx -> <empty>

Rule 82 callfuncparx -> , callfuncpar

Rule 83 datatype -> VOID

Rule 84 datatype -> D_BOOL

Rule 85 datatype -> D_STRING

Rule 86 datatype -> D_FLOAT

Rule 87 datatype -> D_INT

Rule 88 check_program -> <empty>

Rule 89 store_init_quad -> <empty>

Rule 90 store_funcm -> <empty>

Rule 91 store_funcv -> <empty>

Rule 92 store_params -> <empty>

Rule 93 store_local_vars -> <empty>

Rule 94 store_mainv -> <empty>

Rule 95 store_op -> <empty>

Rule 96 store_oper -> <empty>

Rule 97 store_const -> <empty>

Rule 98 unload_pv -> <empty>

Rule 99 store_rquad -> <empty>

Rule 100 close_func -> <empty>

Rule 101 store_endif -> <empty>

Rule 102 store_pquad -> <empty>

Rule 103 ver_params -> <empty>

Rule 104 verify_func -> <empty>

Rule 105 add_fstack -> <empty>

Rule 106 end_fstack -> <empty>

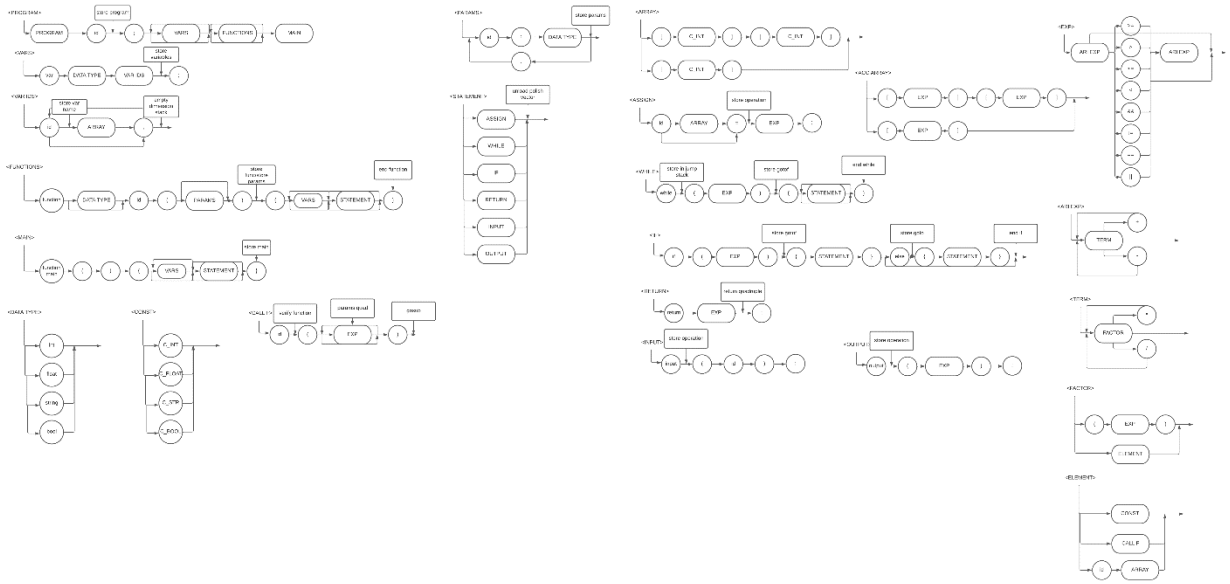
Rule 107 store_gosub -> <empty>

Rule 108 store_gotof -> <empty>

Rule 109 store_jump -> <empty>

Rule 110 end_loop -> <empty>

Descripción de Generación de Código Intermedio



Se puede ver mejor en el diagrama dentro del folder del proyecto.

Esta es la descripción de las acciones semánticas de cada punto:

Store_init_quad: Se guarda el quad inicial tomando el tamaño de la lista de los quads y tomando el nombre de la función sacando la función del diccionario de funciones.

Store_funcm: se toma el scope como main, se crea la función, se añade su quad inicial y se guarda en memoria además de en el diccionario de variables y de funciones.

Store_funcv: Se guarda una función en el diccionario de variables y en el de funciones creando también una dirección para la misma.

Store_params: Se guardan los parámetros que se han guardado en el stack de params.

Store_mainv: Se pone la dirección del goto y se guardan las variables del main tomándolas del stack de local vars

Store_op: se guarda la operación en el stack de operaciones

Store_oper: Se verifica que la variable exista y si es así se guarda en el stack de operandos de los quads

Store_const: Se guarda la constante tanto en el stack de operandos y se le da un address

Unload_pv: se quita todo del polish vector

Store_rquad: Se genera el return quad y se añade el operando de la función, y las operaciones “return” y “endfunc”

Store_endif: Se termina en gotof

Store_pquad: Se genera el quad de parámetros para añadirlos también al stack de operandos y la operación “params” al stack de operadores.

Ver_params: Verifica que el numero de parámetros mandados sea el mismo al numero de los parámetros declarados

Verify_func: Se verifica que la función esté en el diccionario de funciones

Store_gosub: Se añade el operando de la función y el “gosub” a los operadores además de que se añade el return de la función

Store_gotof: Se añade el operador “gotof”

Store_goto: Se añade el operador “goto”

Store_jump: Se añade el quad al stack de jumps

End_loop: Se añade el quad “gotow” al stack de operadores

Descripción de Administración de Memoria:

Para la tabla de funciones se hizo una variable de funciones donde podemos tener los siguientes atributos:

```
class Function:
    name: str
    data_type: str
    returnVar: bool
    initQuad: int
    addr: int
```

Y la tabla de funciones solamente es un diccionario de funciones donde se guarda el nombre de una función como key y el value sería el objeto función.

Para las variables se hizo lo mismo, se declaró una clase variable para almacenar los siguientes atributos:

```
class Variable:
    name: str
    data_type: str
    value: str = "Null"
    addr: int
    dimensions: int
    spaces: int
    scope: str
```

Y la tabla de variables igualmente es un diccionario de variables donde se guarda el nombre de una variable como key y el value seria el objeto variable. Se usaron clases porque usar diccionarios dentro de diccionarios sería más complicado a mi parecer.

Para las direcciones virtuales se hizo la delimitación de los datos como se hablo previamente ordenados entre ints, floats, strings y booleanos y cada uno definido en globales, locales y constantes.

```
# Number of spaces available for each delimitation
area = 100

# Dictionary that calculates the amount of space for each type within a certain scope

territories = {
    # Integers
    "global_int": 0 * area,
    "local_int": 4 * area,
    "constant_int": 8 * area,

    # Floats
    "global_float": 1 * area,
    "local_float": 5 * area,
    "constant_float": 9 * area,

    # Strings
    "global_string": 2 * area,
    "local_string": 6 * area,
    "constant_string": 10 * area,

    # Booleans
    "global_bool": 3 * area,
    "local_bool": 7 * area,
    "constant_bool": 11 * area
}
```

Cada uno con un contador que se va actualizando como se van insertando cada dato (que se hará en el parser en compilación), después el parser genera un diccionario con estos datos para poder trabajar con ello en ejecución, en la memoria de ejecución se declara un diccionario con arreglos con un numero de espacios basados en el numero de datos que el parser desarrollo en sus diccionarios.

```
memory = {
    "global": {
        address["global_int"]: [],
        address["global_float"]: [],
        address["global_string"]: [],
        address["global_bool"]: []
    },
    "local": {
        address["local_int"]: [],
        address["local_float"]: [],
        address["local_string"]: [],
        address["local_bool"]: []
    },
    "constant": {
        address["constant_int"]: [],
        address["constant_float"]: [],
        address["constant_string"]: [],
        address["constant_bool"]: []
    }
}
```

Referencias:

https://ruslanspivak.com/lsbasi-part1/

Writing Compilers and Interpreters: A Software Engineering Approach

https://www.amazon.com/gp/product/193435645X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=193435645X&linkCode=as2&tag=russblo0b-20&linkId=MP4DCXDV6DJMEJBL

Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers)

Compilers: Principles, Techniques, and Tools (2nd Edition)