Diseño de compliadores

Daniel Charua A01017419

13/05/19

Entrega Final

Dr. Victor Manuel de la Cueva

Instrucciones

El compilador es un programa en Python encargado de verificar la 'correctez', mediante el léxico y la semántica, del lenguaje C-. Posteriormente traduce el código a ensamblador para una maquina con la arquitectura MIPS. El código generado se puede correr en una máquina con estas características o bien, en algún simulador como QtSpim.

Ejecución

El Usuario debe de tener instalado Python, en caso de no tenerlo lo pude instalar abriendo una terminal en UNIX y ejecutando el comando:

sudo apt-get install python3.6

Para verificar la instalación ejecutar el comando **python --version**

```
dcharua@dcg:~$ python --version
Python 3.6.3 :: Anaconda, Inc.
```

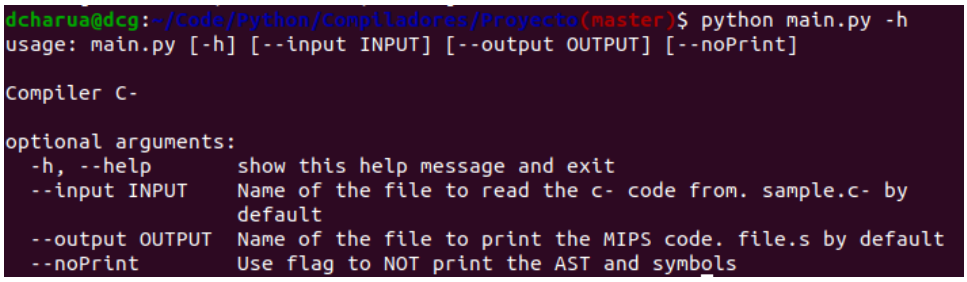
Una vez verificada la instalación se debe de acceder a la carpeta que contiene los archivos del compilador, mediante el comando cd "el path del archivo". Ejemplo: **cd C:\Users\Daniel\Downloads.**

Usar el Programa

Si ya se encuentra en la carpeta, el usuario debe de ingresar el comando **python main.py (argumentos)***

Donde los argumentos pueden ser:

- **--input 'nombre del archivo'**: el nombre del archivo que contiene el código en C- a compilar.
- **--output 'nombre del archivo'**: el nombre del archivo en el cual se gurdara el codigo MIPS
- **--noPrint:** Si se incluye este flag el programa NO imprimirá el AST y la tabla de símbolos
- **-h** despliega la tabla de ayuda para los argumentos: **pyhton main.py -h**



Si no se pasa ningún argumento, **pyhton main.py,** por default el programa corre imprimiendo tanto el AST como la tabla de símbolos, se usa el código definido en sample.c- y se guarda en el archivo file.s

Apéndice

Semantica

Se genera la tabla semántica a partir del AST, donde se ordenan los nodos de acuerdo a su tipo y su scope. Posteriormente de acuerdo a la semántica definida, se verifica la correctez del código.

Se usuaron las siguientes reglas semanticas para verificar el codigo.

Reglas

Las variables y funciones deben de estar definidas se dentro del scope, para ser asignadas y acceder a su valor.

Las funciones deben de regresar el tipo de valor definido en el cuerpo de la función

Para expresiones aritméticas

Int = numeró entro

Int * Int = Int

Int / Int = Int

Int + Int = Int

Int - Int = Int

Int < Int = Int

Int <= Int = Int

Int > Int

int[int] = Int

Expresiones regulares

Se implementó del analizador léxico con expresiones regulares, los siguientes expresiones fueron usadas:

```
(r'[ \s\n\t]+',   TokenType.SPACE),
(r'',             TokenType.SPACE),
(r'#[^\n]*',      TokenType.SPACE),
(r'\$',           TokenType.ENDFILE),
(r'\{',           TokenType.SPECIAL),
(r'\}',           TokenType.SPECIAL),
(r'\[',           TokenType.SPECIAL),
(r'\]',           TokenType.SPECIAL),
(r'\(',           TokenType.SPECIAL),
(r'\)',           TokenType.SPECIAL),
(r'\/\*(\*(?!\/)|[^*])*\*\/', TokenType.COMMENT),
(r';',            TokenType.SPECIAL),
(r',',            TokenType.SPECIAL),
(r'\+',           TokenType.SPECIAL),
(r'-',            TokenType.SPECIAL),
(r'\*',           TokenType.SPECIAL),
(r'/',            TokenType.SPECIAL),
(r'<=',           TokenType.SPECIAL),
(r'<',            TokenType.SPECIAL),
(r'>=',           TokenType.SPECIAL),
(r'>',            TokenType.SPECIAL),
(r'==',           TokenType.SPECIAL),
(r'!=',           TokenType.SPECIAL),
(r'=',            TokenType.SPECIAL),
(r'else',         TokenType.RESERVED),
(r'if',           TokenType.RESERVED),
(r'int',          TokenType.RESERVED),
```

(r'return', TokenType.RESERVED),

(r'void', TokenType.RESERVED),

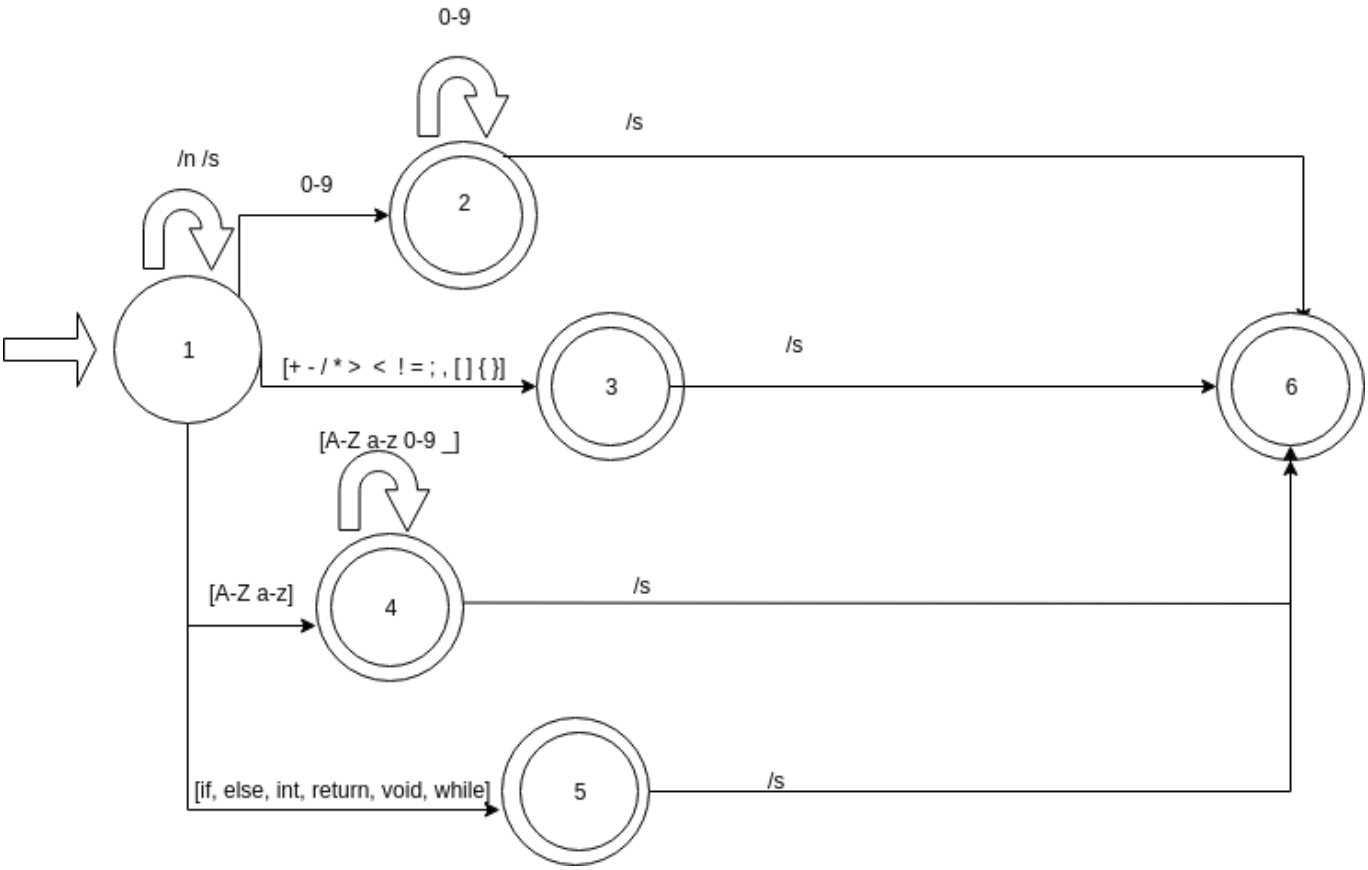
(r'while', TokenType.RESERVED),

(r'[0-9]+[a-zA-Z]+', TokenType.ERROR),

(r'[0-9]+', TokenType.NUM),

(r'[a-zA-Z_][0-9a-zA-Z_]*', TokenType.ID)]

Automata



Gramática:

program -> declaration-list

declaration-list -> declaration {declaration}

declaration -> var-declaration | fun-declaration

var-declaration -> type-specifier [ID ; | ID [NUM] ;]

type-specifier -> int | void

fun-declaration-> type-specifier ID (params) compound-stmt

params-> param-list | void

param-list -> param {, param}

param -> type-specifier [ID | ID [ ]]

compount-stmt -> { local-declarations statement-list }

local-declarations -> empty {var-declaration}

statement-list -> empty {statement}

statement -> expression-stmt | compound-stmt | selection-stmt | iteration-stmt | return-stmt

expression-stmt -> expression ; | ;

selection-stmt -> if (expression) statement | if (expression) statement else statement

iteration-stmt -> while (expression) statement

return-stmt -> return ; | return expression ;

expression -> var = expression | simple-expression

var -> ID | ID [expression]

simple-expression -> additive-expression [relop additive-expression]

relop -> <= | < | > | >= | = | !=

additive-expression -> term {addop term}

addop -> + | -

term -> factor {mulop factor}

mulop -> * | /

factor -> (expression) | var | call | NUM

call -> ID (args)

args -> arg-list | empty

arg-list -> expression {, expression}