Tecnológico de Monterrey

Campus Santa Fe

TC-3048 Diseño de Compiladores

SpanishPL

Alberto Pascal Garza	A01023607
Rodrigo García Mayo	A01024595
Manuel Guadarrama Villarroel	A01020829

Fecha: 6 de Febrero de 2020

Profesor: Edgar Manoatl

¿Cómo utilizar el proyecto?

Clonar o descargar el repositorio que se encuentra en https://github.com/RodrigoGM97/Spanish_PL.
Después de tener la carpeta localmente moverse a ese directorio y ejecutar el siguiente comando: “python3 spanish_main.py”. Por default se ocupa el programa “func.mighty” si se desea cambiar eso entonces en el archivo de spanish_main.py en la línea 10 solos e cambia el nombre del archivo al que se desee utilizar.

```
with open('func.mighty', 'rt', encoding='utf-8') as file:
```

Archivos necesarios

Para poder ejecutar el código, es necesario tener los siguientes archivos (se pueden obtener del github):

- spanish_main.py
    - Contiene el programa principal que lee el archivo para ejecutar los comandos incluidos
- SpanishYacc.py
    - Contiene la gramática en dónde se definen los posibles comandos que se pueden realizar
- SpanishLex.py
    - Contiene el léxico (tokens, palabras reservadas)

Declaración de variables/expresiones

Creación de variables

Para crear una variable, la declaración debe de contener:

1. tipo de variable (num/texto)
2. nombre de variable ([a-zA-Z_][a-zA-Z0-9_]*)
3. Opcional:
    a. asignación de variable (<-)
    b. valor / expresion
        i. texto: "([^\\"]|(\\.))*\"
        ii. num:
            1. float (\d+\.\d*(e-?\d+)?)
            2. int (\d+)
4. ;

Ejemplos:
num hola;
num hola3 <- 0;
num hola1 <- 2 + 2;
texto tex;
texto tex2 <- “hola”;
texto tex3 <- “hola” + “mundo”;

Creación de arreglos

Para crear un arreglo, la declaración debe de contener:

1. tipo de arreglo (num/texto)
2. los tokens ‘[‘ ‘]’
3. nombre del arreglo ([a-zA-Z_][a-zA-Z0-9_]*)

Ejemplos:
texto[] arr;
num[] numeritos;

Asignación de variables

Para asignar un valor a una variable, la variable debe existir y la declaración debe de contener:

1. nombre de la variable
2. el token ‘<-’
3. valor que se le quiere dar a la variable
    a. texto: "([^\\"]|(\\.))*\"
    b. num:
        i. float (\d+\.\d*(e-?\d+)?)

ii. int (\d+)

Ejemplos:
texto tex;
tex <- “hola”;
num nume;
nume <- 1;

Agregar elementos a un arreglo

Para poder agregar elementos a un arreglo, se crea un arreglo y se realiza lo siguiente:

1. nombre de arreglo
2. .
3. agrega
4. (
5. expresión (la expresión tiene que ser acorde al tipo declarado de arreglo)
6.)
7. ;

Ejemplos:
ArrNum.agrega(1);
ArrNum.agrega(1+1);
ArrTexto.agrega(“hola”);
ArrTexto.agrega(“hola”+”mundo”);

Obtener elementos de un arreglo

Para poder obtener el valor de un índice en el arreglo, el arreglo debe existir y se realiza lo siguiente:

1. nombre de arreglo
2. .
3. obtiene
4. (
5. índice o posición del elemento en el arreglo (puede generarse de expresiones numéricas)
6.)
7. ;

Ejemplos:
ArrNum.obtiene(1);
ArrNum.obtiene(0);
num a <- 0;
ArrTexto.obtiene(a);

Eliminar elementos de un arreglo

Para poder eliminar un elemento de un arreglo, el arreglo debe existir y se realiza lo siguiente:

1. nombre de arreglo
2. .
3. alv
4. (
5. índice o posición del elemento en el arreglo (puede generarse de expresiones numéricas)
6.)
7. ;

Ejemplo:

```
arreglo.alv(0);
num a <- 0;
arreglo.alv(a);
arreglo.alv(5 - a);
```

Creación de “si” / “sino”

Para la creación de una condición si, se realiza lo siguiente:

1. si
2. (
3. expresión booleana
4.)
5. {
6. …
7. }

Ejemplo:

```
si (1 noes 2) {
        num a <- 0;
}
si (1 = 2) {
        num a <- 0;
}
si (a > b) {
        num a <- 0;
}
```

Creación de “mientras”

Para la creación de un ciclo mientras, se realiza lo siguiente:

1. mientras
2. (

3. expresión booleana
4.)
5. {
6. …
7. }

Ejemplo:
```
num cont <- 5;
mientras (cont noes 0) {
   cont <- cont - 1;
}
```

Impresión de expresiones y variables

Para poder imprimir una expresión o variable se debe poner lo siguiente:

1. imprime
2. (
3. expresión/nombre de la variable
4.)
5. ;

Ejemplo:
```
imprime(“hola”);
imprime(1+1);
imprime(variable);
imprime(variable + “fin”);
```

Expresiones aceptadas

Se pueden realizar las siguientes expresiones (con el tipo de dato correcto, los strings solo aceptan operador +):

- variable operador variable
- variable operador constante
- constante operador constante

Ejemplo:
```
num a <- 0;
a + 4;
a * a;
1 - 1;
“hola” + “hola”
texto b <- “hola”;
b <- b + “mundo”;
```

Expresiones booleanas aceptadas

Para crear una expresión booleana, se pueden realizar las siguientes comparaciones:

- = (igual)
- noes (diferente)
- y (and)
- o (or)
- > (mayor que)
- < (menor que)
- >= (mayor o igual que)
- <= (menor o igual que)

Ejemplo:
a = 0
b noes 1
1 noes 0 y a noes b
cont > 5
cont <= a

Operadores aceptados

Los operadores que se pueden aceptar son:

- + (suma)
- - (resta)
- * (multiplicación)
- ^ (potencia)
- / (división)

Ejemplo:
a <- 1+1;
b <- 2-1;
c <- 1*4;
d <- 5/3
e <- 2^2

Código de ejemplo

Producto punto de vectores:

```
num[] xVector;
num[] yVector;
num sum <- 0;
num temp <- 0;
num hola;

xVector.agrega(1);
xVector.agrega(2);
xVector.agrega(3);
xVector.agrega(4);

yVector.agrega(5);
yVector.agrega(6);
yVector.agrega(7);
yVector.agrega(8);

num cont <- lon(xVector) - 1;
num tempx <- 0;
num tempy <- 0;
num cont2 <- 5;

mientras(cont >= 0){
    tempx <- xVector.obtiene(cont);
    tempy <- yVector.obtiene(cont);
    sum <- sum + tempx * tempy;
    cont <- cont - 1;
    imprime("hola");
    mientras(cont2 >= 0){
        cont2 <- cont2 - 1;
        imprime("dos mientras");
    }
    cont2 <- 5;
}

imprime(sum);texto n <- "n";
```

Documentación técnica

Existen dos partes importantes en nuestro intérprete, los tokens y las gramáticas. Con ellos podemos crear y estructurar el programa con las declaraciones que nosotros definimos.

Palabras reservadas

- 'si': 'SI'
- 'sino': 'SINO'
- 'mientras': 'MIENTRAS'
- 'imprime': 'IMPRIME'
- 'lon': 'LEN'
- 'err': 'ERR'
- 'y': 'AND'
- 'o': 'OR'
- 'num': 'NUM'
- 'texto': 'TEXTO'
- 'noes' : 'NOTEQUAL'
- 'agrega': 'APPEND'
- 'obtiene': 'GET'
- 'alv' : 'ALV'

Tokens

- num -> define que la variable es un número (int o float)
- texto -> define que la variable es un string
- si -> es la expresión condicional del lenguaje de programación (if)
- sino -> es la expresión que se genera en caso de que la condición original del “si” sea falsa
- mientras -> es la expresión cíclica que se ejecuta siempre que la condición de inicio sea verdadera (while)
- imprime -> es la expresión que se utiliza cuando se desea visualizar una expresión o variable en consola (print)
- lon -> regresa la longitud de un arreglo
- y -> es el operador lógico que une dos expresiones que verifica que las dos expresiones se cumplan
- o -> es el operador lógico que une dos expresiones que verifica que alguna de las dos expresiones se cumplan
- noes -> es el operador que verifica dos expresiones no son iguales
- [ ] -> se utilizan para la declaración de arreglos
- { } -> se utilizan para delimitar secciones del código que pueden ser funciones, ciclos mientras o un si.
- ( ) -> se pueden llegar a utilizar para recibir los parámetros de funciones o declaraciones
- obtiene -> regresar el valor de que contiene un índice de un arreglo
- alv -> quita un valor del arreglo
- agrega -> inserta un nuevo valor al arreglo
- “<-” -> se utiliza para la asignación de variables
- “=” -> es el operador que verifica dos expresiones sean iguales
- ‘+’. ‘-’, ‘/’, ‘*’ -> realizar operaciones matemáticas básicas
- ; -> marca el final de una declaración o expresión

Gramáticas

- Gramática inicial del programa

```
def p_s(p):
    """s : p"""
    pass
```

- Gramática para lectura de líneas en el programa

```
def p_p(p):
    """
    p : stmt p
      | expr ';' p
      | boolexpr ';' p
      |
    """
    pass
```

- Gramática del imprime

```
def p_stmt_print(p):
    """
    stmt : IMPRIME '(' print_arguments ')' ';'
    """
```

- Gramática de los argumentos del imprime

```
def p_print_arguments(p):
    """
    print_arguments : expr ',' print_arguments
                    | boolexpr ',' print_arguments
                    | boolexpr
                    | expr
                    |
    """
```

- Gramática del si

```
def p_stmt_if(p):
    """
    stmt : SI '(' boolexpr ')' '{' ifA p ifB '}'
         | SI '(' boolexpr ')' '{' ifA p ifB '}' SINO '{' ifC p ifB '}' """
def p_ifA(p):
    """
    ifA :
    """
```

```
def p_ifB(p):
    """
    ifB :
    """
def p_ifC(p):
    """
    ifC :
    """
```

- Gramática del mientras

```
def p_stmt_while(p):
    """
    stmt : MIENTRAS '(' boolexpr ')' '{' whileA p whileB '}'
    """
def p_whileA(p):
    """
    whileA :
    """
def p_whileB(p):
    """
    whileB :
    """
```

- Gramática para borrar elementos del arreglo

```
def p_alv(p):
    """
    expr : SYMBOL '.' ALV '(' expr ')'
    """
```

- Gramática de las comparaciones booleanas

```
def p_boolexpr_and(p):
    """
    boolexpr : boolexpr AND boolexpr
             | boolexpr OR boolexpr
    """
def p_boolexpr_comparison(p):
    """
    boolexpr : comparison
    """
def p_comparison_gt(p):
    """
    comparison : comparisonA '>' expr
    """
```

```
def p_comparison_lt(p):
    """
    comparison : comparisonA '<' expr
    """
def p_comparison_eq(p):
    """
    comparison : comparisonA '=' expr
    """
def p_comparison_neq(p):
    """
    comparison : comparisonA NOTEQUAL expr
    """
def p_comparison_le(p):
    """
    comparison : comparisonA LE expr
    """
def p_comparison_ge(p):
    """
    comparison : comparisonA GE expr
    """
def p_comparisonA(p):
    """
    comparisonA : comparison
                | expr
    """
```

- Gramática de las operaciones matemáticas

```
def p_expr_plus(p):
    """
    expr : expr '+' expr
    """
def p_expr_minus(p):
    """
    expr : expr '-' expr
    """
def p_expr_mul(p):
    """
    expr : expr '*' expr
    """
def p_expr_div(p):
    """
    expr : expr '/' expr
    """
```

```
def p_expr_pow(p):
    """
    expr : expr '^' expr
    """
def p_expr_unary_minus(p):
    """
    expr : '-' expr %prec UMINUS
    """
```

- Gramática para obtener longitud de elementos en arreglos

```
def p_get_length(p):
    """
    expr : LEN '(' SYMBOL ')'
    """
```

- Gramática de asignación de variables

```
def p_expr_assign(p):
    """
    expr : SYMBOL ASSIGN expr
    """
```

- Gramática de creación de variables

```
def p_expr_create(p):
    """
    expr : TEXTO SYMBOL ASSIGN expr
         | NUM SYMBOL ASSIGN expr
    """
def p_expr_create_arr(p):
    ''' expr : NUM '[' ']' SYMBOL
             | TEXTO '[' ']' SYMBOL
    '''
```

- Gramática para añadir elementos a arreglos

```
def p_expr_arr_append(p):
    '''expr : SYMBOL '.' APPEND '(' expr ')'
    '''
```

- Gramática para obtener elementos de arreglos por índice

```
def p_expr_arr_get(p):
    '''expr : SYMBOL '.' GET '(' expr ')'
    '''
```

- Gramática para simbolizar vacíos

```
def p_expr_or_empty(p):
    """
    expr_or_empty : expr
```

```
                  |
    """
```

- Gramática de agrupación de expresiones por paréntesis

```
def p_expr_paran(p):
    """
    expr : '(' expr ')'
    """
def p_boolexpr_paran(p):
    """
    boolexpr : '(' boolexpr ')'
    """
```

- Gramática de expresiones extras

```
def p_expr_symbol(p):
    """
    expr : SYMBOL
    """
def p_expr(p):
    """
    expr : INT
         | FLOAT
         | STRING
    """
```