PROCESADORES DEL LENGUAJE

Práctica 2

Analizador del lenguaje
Basic Structs

Grupo 19

JORGE IBÁÑEZ RAMOS 100451330
100451330@alumnos.uc3m.es

JUAN MIGUEL PULGAR ESQUIVEL 100451036
100451036@alumnos.uc3m.es

Mayo de 2023

GRADO EN INGENIERÍA INFORMÁTICA
UC3M

Juan Miguel Pulgar Esquivel 100451036
Jorge Ibáñez Ramos 100451330

ÍNDICE

1. Introducción.

En esta práctica se nos pedía utilizar los conocimientos adquiridos en la práctica 1 de tratamiento léxico, sintáctico y semántico para realizar un compilador en Python que fuese capaz de reconocer y analizar un lenguaje llamado "Basic Structs". Este lenguaje está compuesto de expresiones aritmético-lógicas, registros, vectores simples, funciones y estructuras de control básicas.

2. Gramática.

En nuestra gramática hemos decidido utilizar las siguientes reglas de precedencia:

```
precedence = ( ('left', 'OR'),

        ('left', 'AND'),

        ('left', 'COMPARACION'),

        ('left', '+', '-'),

        ('left', '*', '/'),

        ('right', 'UMINUS',  'UPLUS', 'NOT')
```

Símbolos NO Terminales: **{axioma, statements, statement, expression, declaration, names, parametros }**

Literales: **{ '=', '+', '-', '*', '/', '(', ')', '&', '|', '[', ']', '{', '}', ',', '.', ';', '!' }**

Tokens: **{'NAME', 'OCTAL', 'BINARIO', 'HEXADECIMAL', 'REAL', 'NUMBER', 'NEWLINE', 'COMPARACION', 'CHAR', 'BOOL', 'TYPE', 'MIENTRAS', 'FINMIENTRAS', 'VECTOR', 'LONG', 'SI', 'SINO', 'ENTONCES', 'FINSI', 'REGISTRO', 'AND', 'OR', 'NOT', 'FUNCION', 'DEVOLVER'}**

Cada uno de los tokens anteriores tomará uno o varios valores, siendo:

- NAME: Cualquier cadena de carácteres o números con al menos 1 carácter.
- OCTAL: Cualquier número representado en sistema octal.
- BINARIO: Cualquier número representado en sistema binario .
- HEXADECIMAL: Cualquier número representado en sistema hexadecimal.
- REAL: Cualquier número real.
- NUMBER: Cualquier número entero.
- NEWLINE: Toma el valor de salto de línea `'\n'`.

- COMPARACION: Toma los valores; '==', '<=', '>=', '<', '>'.
- CHAR: Cualquier carácter ya sea en mayuscula o minuscula.
- BOOL: Puede tomar los valores 'cierto' o 'falso'.
- TYPE: Puede tomar los valores 'entero', 'real', 'caracter' o 'booleano'.
- MIENTRAS: Toma el valor 'mientras'.
- FINMIENTRAS: Toma el valor 'finmientras'.
- VECTOR: Toma el valor 'vector'.
- LONG: Toma el valor 'long'.
- SI: Toma el valor 'si'.
- SINO: Toma el valor 'sino'.
- ENTONCES: Toma el valor 'entonces.
- FINSI: Toma el valor 'finsi'.
- REGISTRO: Toma el valor 'registro'.
- AND: Toma el valor 'and'.
- OR: Toma el valor 'or'.
- NOT: Toma el valor 'not'.
- FUNCION: Toma el valor 'funcion'.

Reglas de producción:

axioma ::= statements

statements ::= statements NEWLINE statements

| **statement**

| **statements NEWLINE**

| **NEWLINE**

statement ::= NAME = expression

| **TYPE NAME = expression**

| **TYPE names**

statement ::= NAME NAME

names ::= NAME , names

| NAME

statement ::= REGISTRO NAME { declaration }

statement : NAME . NAME = expression

| NAME . NAME [NUMBER] = expression

declaration ::= TYPE names ; NEWLINE declaration

| TYPE names ; declaration

| TYPE names ;

| NAME NAME ; NEWLINE declaration

| NAME NAME ; declaration

| NAME NAME ;

| VECTOR TYPE NAME [NUMBER] ; NEWLINE declaration

| VECTOR TYPE NAME [NUMBER] ; declaration

| VECTOR TYPE NAME [NUMBER] ;

statement ::= VECTOR TYPE NAME [NUMBER]

statement ::= NAME [NUMBER] = expression

statement ::= SI expression ENTONCES NEWLINE statements FINSI

| SI expression ENTONCES NEWLINE statements SINO NEWLINE statements FINSI

| SI expression NEWLINE ENTONCES NEWLINE statements FINSI

| SI expression NEWLINE ENTONCES NEWLINE statements SINO NEWLINE statements FINSI

statement : MIENTRAS expression NEWLINE statements FINMIENTRAS

statement ::= FUNCION NAME (parametros) : TYPE NEWLINE { statements NEWLINE DEVOLVER expression }

| FUNCION NAME ( ) : TYPE NEWLINE { statements NEWLINE DEVOLVER expression }

parametros ::= TYPE NAME , parametros

| TYPE NAME

| NAME NAME , parametros

| NAME NAME

statement ::= expression

expression ::= expression * expression

| expression / expression

| expression + expression

| expression - expression

| expression COMPARACION expression

| expression AND expression

| expression OR expression

| NOT expression

expression ::= - expression

expression ::= + expression

expression ::= (expression)

expression ::= CHAR | REAL | BOOL | OCTAL | NUMBER | HEXADECIMAL | BINARIO

expression ::= NAME . NAME

| NAME . NAME [NUMBER]

expression ::= NAME . LONG

expression ::= NAME

expression ::= NAME [NUMBER]

3. Breve descripción de la solución.

Nuestro programa final se divide en 3 archivos: '*myLexer.py*', '*myParser.py*' y el programa principal.
El programa 'myLexer' se encarga de realizar las comprobaciones léxicas del programa, es decir, de comprobar si los tokens de entrada son válidos. Además realiza el comando lex.lex que se importa como variable en el parser.
El programa 'myParser' realiza las comprobaciones sintácticas y semánticas del código. Cuenta con una tabla de símbolos y una tabla de registros, además de la gramática y las comprobaciones de tipos y sentencias necesarias para el correcto funcionamiento del programa.
El programa principal importa el parser y realiza el comando 'yacc.parse()' para ejecutarlo.

En nuestra práctica, para almacenar las distintas variables que se utilizan en el programa hemos utilizado una tabla de símbolos. Esta tabla es un diccionario con una entrada para cada variable, siendo su clave el nombre. Cada entrada contiene el tipo de la variable y su valor. Además, si es un vector o un registro, se indicará con un string "vector" o "registro".

Para comprobar si los tipos asignados son correctos, siempre que intentamos declarar o cambiar el valor de una variable, realizamos comprobaciones dentro de las reglas de producción. En el caso de declarar una variable con un valor, comprobamos que el tipo de la declaración sea igual que el tipo del valor y en caso contrario salta un error. Al asignar un valor a una variable ya declarada, comprobamos que el tipo del símbolo almacenado sea igual al tipo de la expresión y en caso contrario saltará un error.

Por otro lado, para comprobar si una variable se ha declarado o no, es decir, para que no se pueda redeclarar y no se pueda utilizar una variable no declarada, utilizamos la tabla de símbolos, es decir, cada vez que se intenta declarar o utilizar, comprueba si está en la tabla de símbolos o no.

Además, para los vectores, debemos comprobar que no se intenta acceder al vector sin utilizar un índice o que se está accediendo a un índice existente. Para ello, al crear el vector se añade una lista a la tabla de símbolos con todos los elementos con su valor por defecto. Cuando se accede al vector, se comprueba que se está utilizando un índice y que éste no supera la longitud del vector - 1.

Para los registros, hemos decidido utilizar diccionarios dentro de un diccionario, es decir, la tabla de registros. Cada entrada de la tabla de registros será el nombre del registro, y contendrá un diccionario con una entrada para cada atributo, de la misma manera que la tabla de símbolos pero sin contener el valor.

Cuando se declara una variable de tipo registro, comprueba que el registro esté declarado y si lo está creará una entrada con el nombre de la variable en la tabla de símbolos, que contendrá un diccionario con los atributos, al principio todos con su valor por defecto. Si se quiere modificar un registro se usará la sintaxis nombre_registro.nombre_atributo = expresión. El programa comprobará en primer lugar si la variable a la que se intenta acceder es un registro, y de serlo comprobará que el atributo exista. Si ambas cosas se cumplen, además de las condiciones de tipo, modificará el atributo del registro.

En las operaciones de control de flujo, se cambia el valor de una variable para que el programa no muestre el valor de cada variable, vector o registro al final de la ejecución.

En la parte opcional, hemos decidido implementar recuperación de errores. Para ello, usamos la estrategia de informar y seguir analizando, de forma que cuando el compilador encuentre un error, imprimirá por pantalla (no en un archivo de texto) el error correspondiente y seguirá analizando en busca de más errores. Si hay un error, no se escribirá nada en el archivo de salida.

4. Archivos de prueba.

Hemos decido realizar un archivo de prueba para cada apartado de funcionalidades de BSL que están especificados en la memoria.

Para comprobar que se pueden declarar sentencias, variables y que se le puede atribuir los tipos correspondientes a estas últimas hemos creado el archivo '*sentencias_variables_tipos.txt*'. En este archivo se declaran variables y estructuras de datos de cada tipo (entero, real, booleana, vector y registro). A estos últimos se les asignan valores y se realiza alguna operación.

El segundo archivo '*reglas_de_tipos.txt*', contiene pruebas para comprobar que las conversiones de tipo se realizan correctamente. Entre ellas probamos a asignar a un entero un valor del tipo 'carácter' (tomará el valor del carácter de 0-255) y también asignar a un real un entero. Además se prueba que en operaciones con más de un valor si no se tienen valores del mismo tipo estos pasarán al más restrictivo para poder realizar la operación. Los valores de tipo 'carácter' pasarán a ser de tipo 'real' o 'entero' dependiendo del otro valor, y los valores de tipo 'entero', podrán pasar a tipo 'real'. Para más comprobaciones, en el archivo, también probamos estas conversiones en elementos de un vector de tipo 'real' y en un registro con 2 atributos, uno de tipo 'entero', y otro de tipo 'real'.

Para comprobar el control de flujo tenemos el archivo '*control_de_flujo.txt*', que contiene las dos estructuras de control del lenguaje, saltos condicionales y bucles condicionales. En concreto el archivo tiene cuatro variables, dos de tipo 'entero' y dos booleanas que servirán para poner las condiciones del bucle y los saltos, además de verse modificadas dependiendo de qué condición se cumpla. El archivo tiene un salto condicional primero y luego un bucle que a su vez tiene dentro un salto.

Por último, debido a que hemos decidido realizar la parte opcional de control de errores, contamos con un archivo '*errores.txt*', que contiene una gran variedad de errores para comprobar el tratamiento de estos. Hemos decidido que los errores se tratarán de manera que se notificará al usuario el error y continuará leyendo en busca de más errores, mostrando por pantalla todos lo que haya.

5. Conclusiones.

En cuanto a las consideraciones para la corrección, debemos comentar que hemos decidido implementar las tablas de símbolos y registros como diccionarios, ya que consideramos que era una forma cómoda de buscar los valores. Al terminar la práctica, nos hemos dado cuenta de que este enfoque puede resultar complejo si se desea ampliar la gramática, por lo que pensamos que podría haber sido mejor utilizar un enfoque basado en clases para almacenar las variables. Además, debido a este enfoque, hemos decidido realizar el análisis semántico dentro del parser, de forma que las comprobaciones se realizan justo antes de escribir el valor en el archivo de salida o saltar el error correspondiente.

En conclusión, en esta práctica hemos podido aplicar los conocimientos adquiridos a lo largo de la asignatura de tratamiento léxico, sintáctico y semántico así como la propia creación de una gramática funcional.

Consideramos que el enfoque de la práctica es muy interesante, ya que nos enseña cómo funcionan los compiladores y cómo se puede crear uno, aunque en este caso hayamos utilizado Python.

En cuanto a la carga de trabajo, consideramos que la longitud y dificultad de la práctica son adecuados, ya que el enunciado ha sido entregado con tiempo suficiente para su realización y el peso de la práctica es considerable, por lo que debe tener cierta dificultad y longitud.