PROCESADORES DEL LENGUAJE

ENTREGA PARCIAL

UC3M

ALEJANDRO MOKOV IVANCHOV (100432094)

SAMUEL FERNÁNDEZ FERNÁNDEZ (100432070)

2023/2024

Introducción **3**
Gramática: **4**
Descripción de la solución **9**
Archivos de prueba **10**
Tests para el análisis léxico: 10
Tests para el análisis sintáctico y semántico 11
Test general 12
Tests de código intermedio 13
Conclusiones **13**

Introducción

Como introducción, nos gustaría exponer la parte léxica de nuestra práctica, con el objetivo de mejorar la comprensión de las partes posteriores.

 # Token de un comentario simple

COM_SIMPLE # Representa comentarios de una sola línea que comienzan con '//'

 # Token de un comentario multilínea

COM_MULTI # Representa comentarios multilínea, comienzan con '/*' y terminan con '*/'

 # Tokens para valores numéricos

ENTERO # Representa números enteros, tanto en decimal, binario, octal y hexadecimal

REAL # Representa números reales, tanto en formato decimal como en notación científica

 # Tokens para operadores lógicos

CONJUNCION # Representa el operador lógico AND '&&'

DISYUNCION # Representa el operador lógico OR '||'

IGUAL # Representa el operador de comparación '=='

 # Tokens para operadores de comparación

MAYORIG # Representa '>='

MENORIG # Representa '<='

 # Token para caracteres y cadenas

CARACTER # Representa un carácter encerrado entre comillas simples

IDENTIFIER # Representa identificadores de variables y funciones

STRING # Representa cadenas de caracteres encerradas entre comillas dobles

Gramática:

/* El programa completo */

S' -> archivo

/* Definición del archivo principal */

archivo -> lineas | lambda

lambda ->

Comentario: El archivo puede estar vacío

/* Secuencia de líneas en el archivo */

lineas -> linea_normal ';' | linea_especial

| linea_normal ';' lineas | linea_especial lineas

Comentario: La diferencia entre línea normal y especial es el ";" al final de las líneas normales y no de las especiales

/* Tipos de líneas de código */

linea_normal -> asignacion | declaracion | llamada_funcion

linea_especial -> COM_MULTI | COM_SIMPLE | funcion | condicional

Comentario: Hemos decidido quitar la regla de expresión en las lineas normales porque vemos que de todas las expresiones solo necesitamos las llamadas a funciones

/* Expresiones dentro de las líneas normales */

expresion -> expresion "+" expresion

| expresion "-" expresion

| expresion "*" expresion

| expresion "/" expresion

| expresion "<" expresion

| expresion ">" expresion

| expresion MAYORIG expresion

| expresion MENORIG expresion

| expresion IGUAL expresion

| expresion DISYUNCION expresion

| expresion CONJUNCION expresion

expresion -> "-" expresion %prec UMINUS

expresion -> "!" expresion

expresion -> "(" expresion ")"

expresion : ENTERO| REAL | TR | FL | NULL

| CARACTER | IDENTIFIER | propiedad | llamada_funcion | objeto_expresion

Comentario: Hemos decidido modificar radicalmente nuestras expresiones porque no habíamos introducido las precedencias y porque nuestras expresiones no reconocían ciertos inputs. De esta manera, nuestras expresiones reconocen todo lo que se pide en la práctica sin tener conflictos al usar las precedencias y así podemos usarlas en los saltos y bucles condicionales, al haber añadido los operadores booleanos en las mismas. Así también hemos suprimido las reglas de operadores unarios y binarios. También hemos integrado los elementos que deben ir dentro de cada expresión como números, caracteres, propiedades, etc.. Por último, hemos añadido las expresiones con paréntesis, puesto que nos faltaban

/* Propiedades y acceso a objetos */

propiedad : IDENTIFIER "[" STRING "]" | IDENTIFIER "." propiedad

| IDENTIFIER "." IDENTIFIER | IDENTIFIER "[" STRING "]" "." IDENTIFIER

Comentario: Hemos hecho cambios en esta parte para poder hacer propiedades anidadas, como hola[“euro].price.spain, o bien, hola.price.spain[“Valencia”]

/* Asignaciones */

asignacion -> IDENTIFIER '=' expresion | IDENTIFIER '=' expresion

Comentario: Hemos decidido quitar IDENTIFIER '=' objeto, puesto que los objetos van en las expresiones

/* Declaraciones */

declaracion -> LET lista_declaraciones | TYPE IDENTIFIER '=' objeto

lista_declaraciones -> contenido_declaracion

| contenido_declaracion "," lista_declaraciones

contenido_declaracion -> IDENTIFIER| IDENTIFIER ":" IDENTIFIER

| IDENTIFIER ":" IDENTIFIER "=" expresion| IDENTIFIER "=" expresion

Comentario: Hemos decidido introducir la lista_declaraciones para hacer más eficiente la recursividad por la derecha del número de variables declaradas. Después, hemos podido simplificar el contenido de las declaraciones al haber agrupado todo en las expresiones. Para las declaraciones de objetos, hemos hecho unos cambios en los objetos que explicaremos más adelante.

/* Estructuras de control y condiciones */

condicional : IF condiciones contenido_if | IF condiciones contenido_if_else contenido_else

| WHILE condiciones_while contenido_while

condiciones : "(" expresion ")"

condiciones_while : "(" expresion ")"

contenido_while : "{" lineas_cond "}"

contenido_if : "{" lineas_cond "}"

contenido_if_else : "{" lineas_cond "}"

contenido_else : ELSE "{" lineas_cond "}"

lineas_cond : linea_cond | linea_cond lineas_cond

linea_cond : linea_normal_cond ";" | linea_especial_cond

linea_normal_cond : llamada_funcion | asignacion | declaracion

linea_especial_cond : COM_MULTI| COM_SIMPLE| condicional

Comentario: Hemos hecho que las condiciones sean expresiones y hemos duplicado nuestro código de líneas quitando la declaración de funciones, puesto que no se debe poder declarar una función dentro de los saltos y bucles condicionales. Además, hemos reestructurado esta parte de la gramática porque así hemos podido implementar el código intermedio para los saltos condicionales, las reglas de producción hacen lo mismo, pero con una estructura diferente, puesto que con estas reglas más factorizadas podemos implementar los cuartetos del código intermedio en los lugares exactos. Con la anterior estructura solo podíamos introducir los cuartetos antes o después de las declaraciones o asignaciones dentro de los saltos condicionales.

/* Definición de funciones */

funcion -> FUNCTION IDENTIFIER '(' argumentos ')' ':' tipo '{' contenido_funcion'}'

contenido_funcion : lineas_fun RETURN expresion ";" | RETURN expresion ";"

lineas_fun : linea_normal_fun ";"| linea_especial_fun

| linea_normal_fun ";" lineas_fun | linea_especial_fun lineas_fun

linea_normal_fun : llamada_funcion| asignacion | declaracion

linea_especial_fun : COM_MULTI | COM_SIMPLE | condicional

argumentos -> argumento | argumento ',' argumentos | lambda

argumento -> IDENTIFIER ':' tipo

Comentario: Hemos hecho un contenido específico de las funciones, puesto que en la práctica anterior habíamos cometido el error de poner returns en los saltos y bucles condicionales. También hemos duplicado el código de líneas para adaptarlo a las declaraciones de funciones, igual que en los saltos y bucles condicionales

/* Llamadas a funciones y parámetros */

llamada_funcion -> IDENTIFIER '(' parametros_funcion ')'

parametros_funcion -> expresion | expresion ',' parametros_funcion | lambda

Comentario: Hemos integrado las expresiones en los parámetros para hacerlos más eficientes.

/* Definición de objetos y sus componentes */

objeto_expresion -> '{' pares_expresion '}'

pares_expresion -> par_expresion ',' pares_expresion | par_expresion

par_expresion -> clave_expresion ':' expresion

clave_expresion -> IDENTIFIER | STRING

objeto_tipo -> '{' pares_tipo '}'

pares_tipo -> par_tipo ',' pares_tipo | par_tipo

par_tipo -> clave ':' tipo

clave_tipo -> IDENTIFIER | STRING

tipo -> INT| FLOAT| BOOLEAN| CHARACTER| IDENTIFIER

Comentario: Hemos decidido duplicar el código de objetos para hacer que los objetos de las declaraciones sólo puedan tener tipos en los valores de las claves y que los objetos de las expresiones sólo puedan tener expresiones en los valores de sus claves

/* Tipos de datos básicos y definidos */

tipo -> INT | FLOAT | BOOLEAN | CHARACTER | IDENTIFIER

Comentario: Hemos añadido IDENTIFIER a los tipos para los tipos definidos por el usuario (objetos)

Descripción de la solución

En la tabla de símbolos almacenamos las variables declaradas con sus tipos y valores. Esto lo hacemos en forma de diccionario donde la clave es el nombre de la variable y el valor es otro diccionario por el tipo y el valor de la variable. De esta forma se pueden buscar las variables de forma más eficiente. En las declaraciones, comprobamos que la variable no esté declarada. En las asignaciones comprobamos el tipo y valor de la variable para asignarlos o reasignarlos en caso de haberlos especificado en las declaraciones. Si una variable es de tipo objeto, el tipo será el nombre del objeto declarado (procedente de type IDENTIFIER = objeto) y su valor es una lista de diccionarios con el nombre y el valor de cada variable dentro del objeto, comprobando que el objeto haya sido previamente declarado y almacenado en la tabla de registros.

En la tabla de registros almacenamos los objetos declarados y las funciones. Para los objetos usamos el mismo procedimiento explicado en la última frase del párrafo anterior con la diferencia de que almacenamos el nombre y el tipo, en vez del valor, del objeto. Para poder hacer lo mismo con las funciones y poder tener sobrecarga de funciones, es decir, tener funciones con el mismo nombre, pero con diferentes argumentos, hemos decidido hacer que el nombre de la función (clave) en la tabla de registros sea un string compuesto por el nombre de la función, número de argumentos y tipo de los mismos separados por comas. De esta manera hay que buscar en la tabla de registros la función con los argumentos exactos, haciendo posible la sobrecarga

Todo el código relativo al análisis semántico está implementado en un archivo llamada funciones_semantico.py

En cuanto al manejo de errores, hemos implementado una serie de excepciones que saltan cuando hay un error, mostrando un mensaje que indica cuál es el error exacto. Todo ello en el archivo excepciones.py

Respecto al código intermedio, hemos implementado código intermedio para asignaciones, declaraciones y saltos condicionales. Para ello, nos hemos inspirado en los ejemplos de código intermedio del enunciado. Lo que hacemos es generar un archivo llamado codigo_intermedio.out, donde almacenamos los cuartetos correspondientes a cada asignación, declaración o salto condicional. Para cada tipo de código, hemos diseñado una función que implementa los cuartetos correspondientes. Al final de todo hacemos un procesamiento de los cuartetos para adaptarlos a lo que pide el enunciado, en otras palabras, si vemos que algún valor está guardado en un registro temporal, por ejemplo, t1, sustituimos el valor por el nombre del registro temporal.

Archivos de prueba

Tests para el análisis léxico:

 # test_enteros: Queremos probar que nuestro token t_ENTEROS reconoce todos los tipos de números con signo positivo y negativo y calcula bien sus valores. Los resultados obtenidos de este test son los esperados.

 # test_reales: Queremos probar que nuestro token t_REALES reconoce números en notación científica y en notación con punto decimal con signo positivo y negativo y calcula bien sus valores. También queremos probar que reconoce números que empiecen por un punto, sin 0 delante. Los resultados obtenidos de este test son los esperados.

 # test_caracter: Queremos comprobar que se reconocen correctamente los identificadores (sin comillas), strings (con comillas) y los caracteres (un carácter con comillas simples). Los resultados obtenidos de este test son los esperados.

 # test_reservadas: Queremos comprobar que se reconocen las palabras reservadas solo si están escritas en minúsculas. Los resultados obtenidos de este test son los esperados.

Tests para el análisis sintáctico y semántico

 # test_let: Queremos comprobar que nuestra gramática hace bien las declaraciones y asignaciones, usando expresiones en estas últimas. No nos da errores, que es el resultado esperado.

 # test_if: Queremos comprobar que se reconocen correctamente los if, if_else y while. Hemos puesto algunos casos límite como un if_else sin condiciones, el cual da error, y otro if_else donde no especificamos que debe hacer, es decir, sin contenido y otro donde las condiciones no dan un valor booleano, que da error. Estos resultados son esperados, dado que no puede haber condicionales sin condiciones o contenido que ejecutar en caso de que las condiciones se cumplan.

 # test_function: Igual que en el test anterior, queremos probar que las funciones se reconocen bien, además de las llamadas a las funciones. Hemos puesto casos límite también como funciones sin argumentos, o funciones sin contenido. En este caso las funciones sin argumentos no dan error, puesto que puede haber funciones que no necesiten de argumentos para funcionar, igual que esas funciones tampoco necesitan parámetros al ser llamadas. Tenemos errores en los tests de funciones sin tipo, de tipo diferente y las funciones vacías de contenido, lo cual es un resultado esperado

 # test_type: Este test sirve para probar la declaración de tipos con objetos. Hemos puesto un objeto con contenido, uno vacío y otro donde hay un objeto dentro de otro objeto (anidado). El vacío nos da error, puesto que no se pueden declarar objetos vacíos: El anidado no da errores

 # test_propiedades: Hemos hecho un par de tests para comprobar que nuestras propiedades funcionan bien, tanto las simples como las compuestas por más de una propiedad (anidadas) funcionan bien.

 # test_extremo: Este test es una prueba de casos extremos donde:

Hemos tratado un comentario anidado, el cual debería dar error al no ser soportados en el lenguaje

Hemos tratado caracteres y escapes inválidos, lo debería dar error al no ser soportados

Hemos tratado operaciones incompatibles, que de momento para esta entrega parcial deben ser reconocidas

Hemos tratado palabras reservadas como identificadores, lo cual debería dar error

Hemos tratado la sensibilidad a mayúsculas y minúsculas en múltiples declaraciones, lo cual en este caso de test no debería darnos ningún error

Hemos tratado la precisión numérica extrema, no se espera ningún error

Hemos tratado caracteres unicode y cadenas, no se espera ningún error al dar soporte completo a Unicode

Hemos tratado múltiples expresiones condicionales complejas, donde no se espera ningún error

Hemos tratado la complejidad extrema en los condicionales, donde no se espera ningún error.

Hemos tratado grandes números, donde hemos descubierto que la función integrer de python es capaz de tratar como máximo 4300 dígitos (let x = 99999...).

Test general

 # test_general: Es un test diseñado para todos los análisis. Para la parte léxica, queremos comprobar que reconoce correctamente todos los tokens. Para la parte sintáctica, queremos comprobar que reconoce bien declaraciones, funciones, ifs... Para la parte semántica queremos ver como quedan las tablas de símbolos y registros. Los resultados obtenidos de este test son los esperados.

Tests de código intermedio

 # test_codigo_intermedio: Hemos hecho una serie de tests para probar que nuestro código intermedio sale tal y como muestran los ejemplos del enunciado. Hemos hecho tests para asignaciones y declaraciones, saltos condicionales con y sin else. Los resultados son los esperados

Conclusiones

Honestamente, no nos dimos cuenta de la importancia de las precedencias hasta que pudimos resolver los 11 conflictos que tenía nuestra gramática gracias a un buen uso de las mismas.

Nos hubiese gustado encontrar una forma más recursiva de implementar la declaración y asignación de objetos anidados, pero no hemos sabido dar con una solución computacionalmente realizable, es decir, que no tenga una complejidad excesiva

Hemos echado en falta la inclusión de ciertas casuísticas en el enunciado, es decir, la inclusión de cómo debe ser el comportamiento de nuestro parser ante determinados inputs más complejos, aunque hemos recibido las oportunas aclaraciones por parte del profesor.

Sugerimos al profesorado la posibilidad de diseñar las prácticas para su realización individual, puesto que creemos que es un contenido que se puede aprender mejor de forma individualizada.