Universidad Carlos III

Curso Procesadores del Lenguaje 2023-24

MEMORIA PRÁCTICA 3

TITULACIÓN:	**INGENIERÍA INFORMÁTICA**	**Grupo**	**80**
Alumno/a:	**Rubén De Arriba Viejo**	**NIA:**	**100471975**
Alumno/a:	**Ana María Ortega Mateo**	**NIA:**	**100472023**

Tabla contenido

Introducción

Este documento presenta los razonamientos y pasos seguidos para implementar el analizador semántico requerido para la práctica final. Se ha partido de la práctica anterior(P2), y se han realizado modificaciones en la parte sintáctica para simplificar el proceso del analizador semántico. Se han creado tres tablas distintas; tabla de símbolos, tabla de registros y tabla de funciones.

La tabla de símbolos sirve para almacenar la información de las variables de tipo básico. Se almacenará su nombre junto con su tipo y en algunos casos su valor. Estas variables no están tipadas por lo que su tipo podría variar dependiendo de las asignaciones que se realizan en el código.

La tabla de registros sirve para almacenar la información de las variables complejas de tipo objeto. Estas variables están fuertemente tipadas, es decir, no pueden cambiar su tipo.

La tabla de funciones se utiliza para almacenar la información de las funciones definidas en el programa. Se llevan a cabo comprobaciones semánticas, asegurando la coherencia en los tipos de datos y el paso de parámetros, así como en el tipo de valor que devuelve cada función.

Gramática definida

La gramática que hemos definido ha experimentado algunos cambios en comparación con la práctica anterior. La nueva gramática es la siguiente, para facilitar su interpretación, los tokens operadores y delimitadores (tal cual se escriben) se representarán en minúsculas, mientras que los terminales se escribirán en mayúsculas.

```
-GLOBAL(axioma)::= function global
                 | type ; global
                 | let ; global
                 | var_declaration ; global
                 | conditional global
                 | loop global
                 | λ

-FUNCTION::= FUNCTION STR_SIN_COMILLAS ( parametros ) : tipos
{ inscope RETURN valores ; }

- TYPE::= TYPE STR_SIN_COMILLAS = ajson_type

-LET::= LET asignaciones
```

-CONDITIONAL::= *IF (expression) { inscope } adicional*

-LOOP::= *WHILE (expression) { inscope }*

-PARAMETROS::= *STR_SIN_COMILLAS : tipos*
| *STR_SIN_COMILLAS : tipos , parametros*
| **λ**

-TIPOS::= *INT*
| *FLOAT*
| *BOOLEAN*
| *CHARACTER*
| *STR_SIN_COMILLAS*

-INSCOPE::= *type ; inscope*
| *let ; inscope*
| *var_declaration ; inscope*
| *conditional inscope*
| *loop inscope*
| **λ**

-EXPRESSION::= *comparation*
| *comparation && expression*
| *comparation || expression*
| *! expression*

-COMPARATION::= *subsum < comparation*
| *subsum > comparation*
| *subsum >= comparation*
| *subsum <= comparation*
| *subsum == comparation*
| *subsum*

-SUBSUM::= *assignation_var + subsum*
| *assignation_var - subsum*
| *assignation_var*

-ASSIGNATION_VAR::= *var*
| *var * assignation_var*
| *var / assignation_var*

-VAR::= *entero*
| *flotante*
| *parentesis*
| *caracter*
| *identificador*

```
      | booleano
      | valor_nulo
      | funciones
      | unarios
-BOOLEANO::= TR
           |FL

-VALOR_NULO::= NULL

-ENTERO::= INT_VALUE

-CARACTER::= CARACTER

-FLOTANTE::= FLOAT_VALUE

-UNARIOS::= - var
          | + var

-IDENTIFICADOR::= STR_SIN_COMILLAS valores_anidados

-PARENTESIS::= ( expression )

-AJSON_TYPE::= { contents_type }


-ASIGNACIONES::= tipadas
                | no_tipadas

-TIPADAS::=  STR_SIN_COMILLAS : STR_SIN_COMILLAS ,asignaciones
            | STR_SIN_COMILLAS : STR_SIN_COMILLAS = valores
            | STR_SIN_COMILLAS : STR_SIN_COMILLAS

-NO_TIPADAS::=  STR_SIN_COMILLAS , asignaciones
              | STR_SIN_COMILLAS = valores
              | STR_SIN_COMILLAS

-ADICIONAL::= ELSE { inscope }
            | λ

-CONTENTS_TYPE::= key : tipos
                | key : tipos ,
                | key : tipos , contents_type
                | key : ajson_type
                | key : ajson_type ,
                | key : ajson_type ,contents_type

-VALORES::= ajson_value
```

```
          | expression

-KEY::= STR_CON_COMILLAS
      | STR_SIN_COMILLAS

-AJSON_VALUE::= { contents }

-CONTENTS::= key : valores
           | key : valores ,
           | key : valores , contents

-NUMBER ::= INT_VALUE
          | FLOAT_VALUE

-VALORES_ANIDADOS::= [ STR_CON_COMILLAS ] valores_anidados
                   | . STR_SIN_COMILLAS valores_anidados
                   | λ

-EXPRESSION_LIST::= expression
                  | expression , expression_list
                  | ajson_value
                  | ajson value , expression_list


-VAR_DECLARATION::=STR_SIN_COMILLAS valores_anidados = valores
```

• Justificación de cambios en la gramática

Para simplificar la tarea del analizador semántico, hemos separado la regla *var* original en varias reglas de producción más específicas. Esto se debe a que si el análisis llega hasta una de estas reglas, significa que el token es de ese tipo. Anteriormente, *var* contenía todas las expresiones en esta única regla y, por tanto, el analizador semántico tendría que realizar un análisis más complejo para saber el tipo de la variable. Sin embargo, al hacer esta separación de reglas, cada tipo se maneja en reglas separadas, lo que significa que si ha llegado hasta esa regla de producción, la variable es de ese tipo, por lo que se podrá asignar el tipo directamente sin tener que hacer ningún procedimiento complejo y, de esta manera, simplificar la comprobación de tipos.

También se ha modificado la forma en que se manejan las asignaciones en nuestra gramática. Anteriormente, todas las asignaciones se trataban bajo una única regla, lo que complicaba la distinción entre variables simples y variables complejas de tipo objeto. Para facilitar esta tarea, se ha dividido asignaciones en dos reglas separadas; *tipadas* y *no_tipadas*. Se podrán manejar de manera diferente y, por lo tanto, añadir las variables a la tabla correspondiente. Las variables simples se añaden a la tabla de símbolos y las variables complejas (objetos) a la tabla de registros.

Nuestra gramática ha sido diseñada de tal manera que no permite expresiones sueltas. Se ha tomado la decisión de considerar que estas expresiones carecen de relevancia en la gramática y no aportan ninguna funcionalidad al programa. Por lo tanto, hemos optado por no permitir expresiones sueltas en el código.

Por último, se ha decidido mantener las expresiones en reglas separadas por niveles, tal y como se había realizado en la práctica anterior. Al separar estos operadores en reglas y colocar aquellos con mayor precedencia en niveles más profundos, permitimos que el analizador evalúe estas sentencias primero, garantizando así el orden correcto de los operadores, siguiendo la teoría vista en clase. Se ha decidido mantenerlo así ya que, cumple con su función. Además, al tener las expresiones divididas por operadores en reglas separadas, hemos logrado una mayor claridad en el código, facilitando la implementación de la comprobación de tipos y conversiones semánticas, ya que cada regla se encarga de manejar un conjunto específico de operadores y cada operador tiene tipos de datos concretos de entrada y devuelve un tipo específico dependiendo de la entrada (siguiendo la tabla del enunciado).

Descripción de la solución

Para llevar a cabo el análisis semántico, hemos establecido tres tablas distintas que se encargarán de almacenar la información relevante sobre las variables y funciones definidas y asignadas en nuestro programa.

- Tabla de símbolos

La clase “SymbolTable” es nuestra tabla de símbolos, encargada de almacenar las variables de tipo simple. Esta clase tiene dos métodos principales; “add_symbol”, para añadir una nueva entrada a la tabla con el nombre de la variable (si no se encuentra ya almacenada), el tipo, el valor (en caso de que tenga) y scope (nivel), “update_symbol”, al ser variables simples no tipadas pueden cambiar de tipo, por lo que si la variable se encuentra almacenada en la tabla de símbolos se actualizará su tipo y valor y “get_type”, que se utiliza para obtener el tipo y el valor de las variables simples.

Se pueden definir variables sin ningún valor, simplemente se almacenará el nombre y su tipo y valor se establecerán a “NULL” y None respectivamente. Se ha decidido así ya que la variable existe porque se ha definido pero no se ha almacenado ningún valor en ella. También sucederá esto cuando hay algún tipo de asignación no válida, es decir, cuando se hace una operación con dos tipos no válidos u operaciones cuyo resultado carezca de tipo y valor. Se ha decidido así ya que, aunque la asignación no sea correcta, se está asignando una variable, por lo que esta se añadirá a la tabla de símbolos, pero, como los tipos no son válidos y no se puede asignar nada adecuadamente, su tipo será “NULL” y su valor None como si no hubiera sido asignado a nada. Permitiendo al usuario poder asignar un valor a dicha variable posteriormente sin definirla de nuevo, evitando el error de variable no definida. Se podrá

también, ir almacenando o cambiando los valores y tipos de las variables que se van definiendo y asignando, al ser variables débilmente tipadas. Su contenido se actualiza en *var_declaration* en caso de que exista en la tabla de símbolos.

Como se ha explicado anteriormente, se ha realizado una separación de reglas en el tipo de las variables (*var*) de las expresiones, por lo que si llega a unas de esas reglas significa que es de ese tipo, por lo que se asigna a p[0] una lista con el valor de la variable (p[1]) y el tipo de variable que es; entero, flotante, caracter, booleano, nulo o identificador según la regla a la que haya llegado. En caso de que sea una expresión, se van almacenando los nuevos tipos que se generan tras cada operación, comprobando que cumplen con los tipos permitidos por cada operador y asignando el tipo en función del valor devuelto por ese operador en caso de que la operación se haya podido realizar. Si la operación ocurre entre dos variables de diferente tipo pero que el operador permite, se realiza la conversión correspondiente (siguiendo las normas del enunciado) para que se pueda realizar dicha operación. Los caracteres se transforman a números con *ord()* (transforma a entero) y en caso de que no tengan ninguna operación con una variable de entero o flotante, volverían a ser caracteres (ejemplo en suma y resta). Al hacer operaciones entre enteros y flotantes, directamente se convierte al tipo más restrictivo (flotante), simplemente definimos que si alguna de las dos variables es flotante el tipo devuelto será flotante. En el caso de la división, si el denominador es mayor que el numerador, se establecerá su tipo como flotante, ya que el número devuelto por esa división no es un entero independientemente que ambos números lo sean (Ej: 9/10). Si alguna de las variables es una función se comprobará que el tipo es correcto pero el valor será None, ya que, al no generarse el código de la función, no puede obtenerse un valor en la operación.

Para actualizar el valor de una variable cuando se hace una asignación sin definir (*var_declaration*), se comprueba primero que el nombre de la variable exista para poder llamar al método update_symbol de la tabla. Si se quiere añadir una variable cuyo nombre ya existe, se ha considerado que salté un error para avisar al usuario de que está volviendo a definir una variable ya definida y no se hará ninguna modificación en ella, por lo que solo se permite añadir en la tabla de símbolos variables con un id (nombre) único para así evitar y restringir las alteraciones y modificaciones en nuestra tabla.

- Tabla de registros

La clase “RegisterTable” es nuestra tabla de registros, encargada de almacenar variables de tipo complejo (objetos). La clase tiene un método para poder añadir los tipos de las variables de tipo complejo (“add_type”), en caso de que alguno de los tipos definidos no sea coherente, se descartarán todos los tipos que se hayan añadido en esa regla (“apply_coherence”). Además, se permite añadir valor a los objetos, en caso de que se cumpla con el tipo tanto para el tipo general como para cada uno de los tipos internos (“check_type”), usando la función “add_register”. También tiene el método “obtener_puntero”, para tener acceso al registro en el que se quieren asignar los valores, en caso de que se pueda. La función “none_dict” se utiliza para, en caso de declarar un objeto sin asignarle ningún valor, asignar un diccionario con todos los atributos asignados a [None, tipo], siendo tipo el tipo real de dicha variable. Por

ejemplo, la declaración sin asignación del tipo {a: int, b: {c: boolean}} en la variable "objeto" quedaría de la forma: objeto = {a: [None, "INT"], b: {c: [None, "BOOLEAN"]}}.

Hay dos diccionarios y una lista para almacenar la información en la tabla de registros; un diccionario para guardar los distintos tipos de variables de la tabla de registros (self.types), una lista para almacenar los valores que se van asignando a cada objeto (self.registers) y por último, un diccionario temporal, para ir añadiendo todos los tipos anidados dentro de un tipo, y, en caso de que alguno de estos falle, se descartará toda la copia, descartando en sí todo el tipo y no se añadirá a nuestra tabla de tipos (self.copy). Esto se ha hecho para evitar errores de coherencia en los objetos y que si no es válido en su completitud, no será una tipo complejo válido y no se añadirá.

Para definir una variable de tipo complejo es necesario primero declarar los tipos de los que va a estar compuesto. Los tipos se añaden utilizando el método "add_types". Este método comprueba que el tipo no se haya añadido anteriormente y que el tipo no sea uno de los tipos básicos del lenguaje. Si se cumplen las condiciones, se van añadiendo los tipos de forma recursiva al diccionario self.copy. Si en algún momento del algoritmo se llega a algún tipo que no está definido correctamente, se detendrá el algoritmo, se pondrá la variable self.coherence a False. Posteriormente, se usa el método "apply_coherence" se mira si self.coherence es True o False y, en el caso de True, se añaden los tipos a self.types; en el caso de False, se descartan los tipos añadidos, copiando el valor de self.types, y se cambia self.coherence a True. Esto se hace para que en caso de fallar, los tipos más internos se mantendrían definidos a pesar del fallo producido, por lo que es necesaria una copia de los tipos para mantener los tipos anteriores sin tipos erróneos.

En el caso de que existan diccionarios anidados dentro de la definición de un tipo, por ejemplo type A {b: {c: int}}, se generarán dos tipos: el tipo A, cuya definición es {b: Ab}, y el tipo Ab, cuya definición es {c: int}. De esta forma, definimos los nombres de las variables complejas dentro de los objetos y atribuimos un identificador único a cada una de ellas.

Las variables complejas que incluyen anidaciones en nuestro programa, es decir, ajson anidados, se ha decidido que se almacenen como el par de valores [valor, "DICT"]. Se ha decidido de esta manera porque, ya que tanto los símbolos como los valores dentro de los registros tienen la estructura [valor, tipo], así también los valores de los registros tienen asignado un tipo específico. De esta manera evitamos utilizar estructuras complejas de python para realizar comprobaciones (isinstance o try-except), comprobando directamente si es de tipo "DICT", lo que indica que es un ajson de una forma mucho más simple. Como el tipo "DICT" no es ningún tipo definido en el enunciado de la práctica, se utiliza la función "quitar_tipo" para que en la impresión de tipos de las variables complejas se omita.

Una vez ya definido el tipo, se puede añadir los valores. Antes de añadir la variable a la tabla de registros, como debe ser tipada, se comprueba que el tipo del valor coincida con el tipo definido del objeto, sino no se añadirá ninguno de los valores debido a que a una variable compleja solo se le pueden asignar valores compatibles al tipo establecido. La lógica de

asignaciones con let se hace para asignar valores a todo un objeto, incluido los objetos anidados de su interior, se añadirá con el método “add_register” todos los valores que se van asignando, en caso de que alguna no concuerde no se le asignará el valor a esa variable. La comprobación de tipo se realiza gracias a la función “check_types”. Se ha considerado que, en caso de que esté mal definido algún valor, se descartará la declaración y asignación al ser variables complejas más restrictivas en comparación con las simples.

También se puede modificar los valores dentro de una variable compleja. Es decir, si se accede a un atributo dentro de un objeto se puede modificar su valor siempre y cuando la estructura del tipo se respete. Esto se hace comprobando, si es un valor de tipo entero, flotante, caracter o booleano que la asignación sea de este tipo y, si es otro objeto, que el tipo de ese objeto sea correcto. De esta forma, si se modifica la variable nombre[“atributo”], el tipo de esta debe mantenerse debido a que estas variables están fuertemente tipadas.

Los registros también pueden ser utilizados en las expresiones para realizar operaciones, sin embargo, se deberá usar un atributo con un tipo compatible con la operación que se desea realizar. Esto se hace accediendo de forma recursiva al valor de dicho atributo usando la función “obtener_puntero”, un método recursivo que accede al valor de una variable compleja, como si de un puntero se tratase. Una vez accedido el valor se procederá a hacer las comprobaciones de tipo igual que si fuese un símbolo.

Para que el analizador semántico no tenga problemas a la hora de asignar o definir variables, se ha decidido que los nombres de variables utilizados para definir símbolos no se puedan utilizar para definir registros y viceversa. Esto restringe que se puedan añadir dos variables con el mismo identificador y en el mismo scope en las dos tablas y hace que las asignaciones en *var_declaration* sean más simples de gestionar.

- Tabla de funciones

La clase “FunctionTable” es nuestra tabla de funciones, encargada de almacenar las funciones de nuestro programa. Se ha decidido crear una tabla adicional para las funciones para que cada tabla gestione diferentes aspectos del programa. Esta tabla consta de varios métodos: “add_function”, que permite añadir una función con su nombre, el tipo de los parámetros y el tipo devuelto; “new_function”, que comprueba que se pueda añadir la función, teniendo en cuenta los tipos y número de parámetros así como su nombre; y “check_parameters”, que se encarga de comprobar que los parámetros que se le está pasando a la función son los adecuados en función del tipo correspondiente, en caso de que se le pase un objeto, se comprueba su tipo, utilizando los tipos definidos en la tabla de registros.

Cuando se define una función en el programa, usando la regla de producción function, se añade la función a la tabla de funciones, con su nombre, el tipo que devuelve y una lista con cada uno de los tipos de los parámetros de la función. Se ha decidido pasar los tipos de la tabla de registros para poder comprobar que los tipos de los parámetros y el tipo en sí de la función sean tipos válidos, teniendo en cuenta las variables complejas.

Cuando se llama a una función en una expresión, primero se comprueban los parámetros que se están pasando a la función y si estos son válidos. En caso de que los valores de los parámetros correspondan con los permitidos por la función, se devolverá como valor None, ya que las funciones no devuelven valor al no ser necesario para esta práctica, y se devolverá el tipo que devuelve la función. Si los parámetros no son válidos, se asigna [None, "NULL"] a p[0], indicando que no se ha guardado nada y la llamada a la función ha fallado.

Para definir el tipo de una función, se considera el tipo asignado al momento de su declaración. No se tiene en cuenta el valor que retorna ni si coincide con el tipo declarado de la función, ya que el tipo de la función se determina durante su declaración. Esto permite almacenar la función sin importar el valor de retorno y, al no conocer el valor exacto que devuelve, no lo hemos considerado necesario.

Para manejar los parámetros correctamente, suponiendo el parámetro "a" propio de la función y una variable simple "a" fuera de la función ,estas no se deben confundir, hemos creado un atributo en la tabla de registros y símbolos que es scope. Tampoco se debería confundir cualquier definición de una variable dentro de una función con una de fuera. El scope hace referencia al "nivel" de las variables en el programa, para determinar en qué función se encuentran y a cuál pertenecen. El manejo del nivel de las variables se maneja con la variable global "self.in_function" que se inicializa con el valor de False, indicando que no se encuentra dentro de una función. Cuando llega a la regla de producción de parámetros esta variable global se modifica a True, ya que cuando llega a esta regla de producción significa que se ha alcanzado la definición de una función. Por ello, se ha creado otra variable global (self.next_scope) que tiene el recuento de las funciones que se van creando y, por tanto, los niveles presentes en el programa, es decir, el nivel 0 corresponde a todas las variables fuera de funciones y el resto hacen referencia a cada función declarada, siguiendo el orden en el que son definidas. El scope se incrementa en uno indicando que ha terminado de definir la función. Todas las reglas de producción accedidas, tanto las declaraciones como asignaciones de variables complejas o simples, se habrán almacenado con el scope correspondiente a dicha función. Es decir, las variables correspondientes a la primera función definida tendrán el scope 1, las variables correspondientes a la segunda al 2 y así sucesivamente.

- Bucles y condicionales

En los condicionales y los bucles, se ha realizado una comprobación de la condición de cada uno de ellos. La condición sólo puede ser de tipo booleana, en cualquier otro caso saltará un error. Solo se ha realizado esta comprobación de tipos de la condición pero no hace ningún salto condicional ya que, por no ser una de las premisas principales de la práctica, se ha decidido no realizar control de flujo. Se notificará de los errores, pero el código seguirá y hará la lógica interna, independientemente de si la condición es verdadera o falsa.

- Manejo de errores

Se ha hecho un tratamiento de errores en el que el analizador sintáctico alertará de cada error encontrado y lo maneja de una forma determinada. Los errores que se han gestionado son:

- Si la variable no existe en la tabla de símbolos ni en la de registros.
- Si el tipo de variable compleja que se quiere asignar no coincide con el tipo en la tabla de registros.
- Si la variable no se encuentra registrada al querer actualizarla en la tabla de símbolos o registros.
- Si la condición de los bucles y condicionales no es booleana.
- Si la llamada a una función no es correcta, porque los parámetros no cumplen con los tipos adecuados o no existe en la tabla de funciones.
- Si el identificador al que se desea acceder en la tabla de símbolos en una expresión no existe.
- Si los tipos dentro de los parámetros de una función no son correctos o el tipo devuelto por la función no lo es.

En los operadores como anteriormente se ha explicado, cualquier problema de asignación de las variables simples y funciones se han solucionado asignando a éstas el valor de None y tipo “NULL”.

- Si en los operadores lógicos las variables de entrada no son de tipo booleano.
- Si en los operadores de comparación (menos ==), alguna de las variables de entrada es booleana.
- Si en los operadores de suma y resta, alguna de las variables de entrada es booleana. También saldrá como error si se intenta realizar la resta entre dos caracteres y el positivo es menos que el negativo, esto no será posible ya que no existe ningún índice en la tabla ASCII negativo.
- Si en los operadores de multiplicación y división las variables de entrada son alguna booleana.

Batería de pruebas

Las pruebas se han ido dividiendo según la tabla que se quiere evaluar y según las distintas comprobaciones semánticas y funciones de cada tabla.

- Tabla de símbolos

TEST1.TXT →Este test sirve para poder comprobar que se definen y asignan correctamente las variables simples en nuestra tabla de símbolos. Se realiza las siguientes comprobaciones:

- Definir variables sin asignarle ningún tipo de valor. Se espera que se añadan en la tabla de símbolos con tipo “NULL” y valor None.
- Asignar variables, anteriormente definidas y comprobar que se almacenan en nuestra tabla de símbolos correctamente.
- Utilizar identificadores en las expresiones y comprobar que utiliza correctamente sus valores y tipos.

- Asignar valores con expresiones, utilizando los cuatro tipos de operadores(aritméticos, booleanos, de comparación y los unarios;de signo). Se realizan expresiones que hemos determinado que deberían guardar algún valor y hemos comprobado que la precedencia de los operadores se cumple y almacena en la tabla de símbolos el tipo y valor correctamente. Se ha ido comprobando operador cada operador. En el caso de la conversión se pasa primero de carácter a entero(tipo entero) y de entero a flotante, en el caso que sea una operación entre carácter y flotante.
- Se puede comprobar que permite reasignar variables, cambiando los tipos, y si se puede el valor de las variables ya definidas.

TEST2.TXT →Este test sirve para comprobar que nuestro analizador no permite redeclarar variables ya declaradas en la tabla de símbolos y no permite operaciones entre variables de tipos que el operador no soporta como variables de entrada o combinación. También sirve para comprobar que dos variables no se pueden llamar igual. Se comprueba:

- Operadores +,- : Solo permiten caracteres, flotantes o enteros, comprobamos que con booleanos el resultado es null(tipo) y valor none, es decir, no se asigna y salta el error correspondiente.
- Operadores * y /: Solo permiten flotantes o enteros, comprobamos que con booleanos el resultado es null(tipo) y valor none, es decir, no se asigna y salta el error correspondiente. Los caracteres al poder convertirse no darán error.
- Comparadores <,>,<= o >=: Solo permiten caracteres, flotantes o enteros, comprobamos que con booleanos el resultado es null(tipo) y valor none, es decir, no se asigna y salta el error correspondiente. Los caracteres se comparan pasandolos a ord(), en la lógica del código, por ello, se pueden hacer operaciones como mayor o menor.
- IgualIgual: Permite cualquier tipo de entrada, aunque la combinación de booleanos con cualquier otro tipo no es posible, por lo que se comprueba que da error.
- Comparadores lógicos &&, || y !: Solo permiten como variable de entrada un booleano, por lo que se prueban otros tipos de variables de entrada y se comprueba que salta el error correspondiente.

- Tabla de registros

TEST3.TXT →Este test sirve para poder comprobar que se definen y asignan correctamente las variables compuestas en nuestra tabla de registros. Se realizan las comprobaciones semánticas con una variable objeto, con muchas anidaciones en su interior y un ejemplo considerado bastante completo y que abarca todas las situaciones. Se realiza las siguientes comprobaciones:

- Declarar los tipos de la variable compleja. Se comprueba que se almacenen correctamente tanto las anidaciones de nombres como el tipo de cada atributo definido.
- Se le asigna valores a todo ese objeto, valores concordes al tipo para comprobar que se añaden correctamente.
- Se accede a uno de los atributos específicos de la variable compleja y se comprueba que se realiza la modificación correctamente.

- Se asigna a una variable el tipo del atributo complejo, esta variable se puede comprobar cómo se almacena en nuestra tabla de registros.
- Se inicializan los valores de esta última variable, para comprobar que se almacenan correctamente y posteriormente acceder a uno de sus atributos y cambiarle el valor pudiendo comprobar que se realiza la modificación correctamente.
- Comprobación de que no hay problemas con el tipo DICT, al haberlo especificado nosotros como un tipo de los ajson.
- Anidaciones con acceso con salto para comprobar que se accede correctamente. Es decir, un ejemplo sería un acceso con [“as ada”].

Este test nos ha servido para evaluar el correcto funcionamiento de la comprobación de tipos y asignación de valores(si se puede) en nuestra tabla de registros. Se comprueba que se cumple todo lo permitido por este tipo de variables.

TEST4.TXT →Este test sirve para comprobar que nuestro analizador no permite cambiar el tipo a las variables de tipo complejo, que debe estar inicializada la variable compleja para poder asignar valores y otros tipos de errores considerados. Se prueba:

- No se permite hacer una asignación si no se encuentra en la tabla de registros el tipo a asignar.
- Se tiene que inicializar los valores enteros de la variable para poder acceder a los atributos.
- Si hay alguna asignación en toda la variable compleja que esté mal tipada no se agrega esta en la tabla de registros.
- Si se intenta modificar un atributo complejo mal tipado, no se realizará ningún cambio.
- Si se accede a un atributo no existente, saltará un error.

- Tabla de funciones

TEST5.TXT →Este test sirve para poder comprobar que se definen y asignan correctamente las funciones en nuestra tabla de funciones. Se realiza las siguientes comprobaciones:

- Se declaran distintas funciones, cada una con un tipo distinto.
- Se prueban a realizar operaciones con los operadores para comprobar que se devuelve el tipo esperado(valor no ya que no tienen).
- Se declara una variable que existe en la tabla de símbolos dentro de una función para comprobar que no salta el error.

TEST6.TXT →Este test es para comprobar que nuestro analizador maneja los errores de las funciones correctamente. Los errores comprobados son los siguientes:

- No se puede redeclarar una función dos veces con el mismo número de parámetros.
- Parámetros no permitidos.
- Tipo de función no permitida.
- Operaciones con funciones de tipos no válidos por cada operador.
- Si se redefine dos veces una variable dentro de una función.

- Comprobación de condiciones

TEST7.TXT→Este test es para realizar las comprobaciones de las condiciones de los bucles y condicionales. Se prueban tanto condiciones válidas como no válidas, para comprobar que nuestro analizador detecta correctamente los errores y que sabe detecta correctamente lo que devuelve cada condición, es decir, si es falsa(advertirá de que no se cumple pero no le hemos dado ninguna funcionalidad más) o si es verdadera.

Conclusiones

Esta práctica nos ha servido para poder generar nuestro propio analizador semántico y realizar las comprobaciones de tipos y el análisis según lo aprendido en teoría y aspectos que hemos considerado. Al partir de la gramática ya hecha en la práctica anterior con ligeros cambios para simplificar la tarea del analizador, nos ha resultado mucho más cómodo que si tuviéramos que partir de cero. Es importante tener una gramática bien definida y realizar cambios que simplifiquen el trabajo del analizador semántico para evitar futuros problemas y disminuir el trabajo de este analizador.

Al no haber una forma precisa en la que construir un analizador semántico, a medida que se iba realizando, se han realizado muchas modificaciones debido a que o lo estábamos haciendo mal o se podían mejorar. Al ser una práctica bastante amplia y con un alto grado de libertad a la hora de implementar el analizador, consideramos que se podría seguir perfeccionando y probablemente existan implementaciones más simples para realizar este análisis.

Al hacer este analizador en python y no poder acceder directamente a direcciones de memoria utilizando por ejemplo punteros, ventaja que tienen lenguajes como C, se ha tenido que implementar métodos más complejos y recursivos para poder acceder a variables determinadas.

La fecha de entrega de la práctica no ha ayudado mucho, debido a que, teniendo a la vez que tener que estudiar para los exámenes finales de todas las asignaturas, no se ha podido dedicar la atención necesaria debido a la carga de tareas en general y consideramos que se hubiera llegado a mejores resultados si la fecha de entrega hubiera sido antes (problema nuestro que se nos haya juntado todo por no anticiparnos, también consideramos).