Procesadores del lenguaje

PRÁCTICA 3: AJS LÉXICO Y SEMÁNTICO

30/05/2024

Belén Gómez Arnaldo 100472037

Ignacio Fernández Cañedo 100471955

Índice

Introducción

En esta práctica se incluye el análisis final léxico, sintáctico y semántico, además de las explicaciones relacionadas con el último analizador. En primer lugar se va a incluir la gramática final, así como los cambios que se han realizado sobre esta respecto a la entrega anterior. Después se incluye una breve descripción de la solución que explica cómo se han utilizado las distintas tablas. Posteriormente hay una explicación más detallada sobre el análisis semántico. Finalmente se analizan los test que se han empleado para verificar el correcto funcionamiento del código así como el tratamiento de errores.

Gramática

La gramática final es esta:

```
Unset
 precedence = (
        ('right', 'NOT'),
        ('left', 'AND', 'OR'),
        ('left', 'LE', 'LT', 'GE', 'GT', 'EQ'),
        ('left', 'SUMA', 'RESTA'),
        ('left', 'MUL', 'DIV'),
        ('left', 'USUMA', 'URESTA'),
    )



program :  statement
        | empty

statement : content SEMICOLON
          | content SEMICOLON statement
          | noSM statement
          | noSM
```

```
content : declaration
        | assignment
        | definicion_ajson

noSM : function
     | condition
     | loop
entero : ENTERO

decimal : DECIMAL

num : entero
     | decimal

bool : TR | FL

declaration : LET id

id : var
   | var COMA id
   | var IGUAL expr
   | var IGUAL expr COMA id

var : CSINCOMILLAS
    | CSINCOMILLAS PUNTOS tipo

tipo_ajson : CSINCOMILLAS

tipo : INT
     | FLOAT
     | CHARACTER
     | BOOLEAN
```

```
     | tipo_ajson

assignment : var IGUAL expr
           | punto_valor IGUAL expr
           | corchete IGUAL expr


variable : CSINCOMILLAS

cadena : CARACTER

parentesis : LPARENT expr RPARENT

signos : SUMA expr %prec USUMA
       | RESTA expr %prec URESTA

expr : operacion
     | num
     | bool
     | NULL
     | variable
     | signos
     | cadena
     | ajson
     | parentesis
     | pc
     | functioncall

operacion : aritmetica
          | binaria
          | comparation
```

```
aritmetica : expr SUMA expr
           | expr RESTA expr
           | expr MUL expr %prec MUL
           | expr DIV expr %prec DIV


binaria : expr AND expr
        | expr OR expr
        | NOT expr

comparation : expr LE expr
            | expr LT expr
            | expr GE expr
            | expr GT expr
            | expr EQ expr

definicion_ajson : TYPE CSINCOMILLAS IGUAL ajson_t

ajson_t : LBRACKET object_t RBRACKET

object_t : pair_t COMA object_t
         | pair_t COMA
         | pair_t

pair_t : clave PUNTOS tipo
       | clave PUNTOS ajson_t

clave : CCOMILLAS
      | CSINCOMILLAS

ajson : LBRACKET object RBRACKET
```

```
object : pair COMA object
       | pair COMA
       | pair

pair : clave PUNTOS expr

pc : punto_valor
     | corchete

punto_valor : punto1
            | punto2
            | punto_corchete

punto1 : CSINCOMILLAS PUNTO CSINCOMILLAS

punto2 : CSINCOMILLAS PUNTO punto_valor

punto_corchete : CSINCOMILLAS PUNTO corchete

corchete : CSINCOMILLAS LCORCHETE CCOMILLAS RCORCHETE recur_corchete

recur_corchete : LCORCHETE CCOMILLAS RCORCHETE recur_corchete
               | empty
               | PUNTO CSINCOMILLAS
               | PUNTO punto_valor

condition : IF LPARENT expr RPARENT LBRACKET statement RBRACKET
             | IF LPARENT expr RPARENT LBRACKET statement RBRACKET ELSE
LBRACKET statement RBRACKET

loop : WHILE LPARENT expr RPARENT LBRACKET statement RBRACKET
```

```
function : function_no_args
         | function_args



function_args : FUNCTION CSINCOMILLAS LPARENT arg_list RPARENT PUNTOS tipo
LBRACKET statement RETURN expr SEMICOLON RBRACKET
              | FUNCTION CSINCOMILLAS LPARENT arg_list RPARENT PUNTOS tipo
LBRACKET RETURN expr SEMICOLON RBRACKET
function_no_args  : FUNCTION CSINCOMILLAS LPARENT RPARENT PUNTOS tipo
LBRACKET statement RETURN expr SEMICOLON RBRACKET
          | FUNCTION CSINCOMILLAS LPARENT RPARENT PUNTOS tipo LBRACKET RETURN
expr SEMICOLON RBRACKET


arg_list : CSINCOMILLAS PUNTOS tipo
         | CSINCOMILLAS PUNTOS tipo COMA arg_list


functioncall : CSINCOMILLAS LPARENT RPARENT
             | CSINCOMILLAS LPARENT argumentos RPARENT


argumentos : expr
           | expr COMA argumentos
empty :
```

Se han realizado varios cambios en la gramática para poder realizar el analizador semántico. También se ha modificado el orden de la precedencia para hacer primero las operaciones aritméticas que las de comparación.

Principalmente se han separado producciones de reglas de la misma longitud en reglas distintas, así se consigue diferenciar de forma más específica los distintos casos posibles.

Los números se han separado en reglas distintas, enteros y decimales, de esta forma se puede guardar el tipo adecuadamente. En las expresiones se han creado nuevos no terminales como variables, signos, cadena, o paréntesis para poder acceder al valor y tipo y guardarlos. Las funciones también se han separado en funciones con

argumentos y sin argumentos, ya que también tenían la misma longitud. Las reglas relacionadas con los valores en notación punto y corchete también se han modificado para poder tratarlas en el semántico.

En resumen, la gramática en su mayoría permanece igual que en la anterior entrega. Solo se ha modificado en aquellos casos en los que las generaciones de la misma regla tienen la misma longitud, ya que así el semántico puede tratar adecuadamente los distintos casos.

Breve descripción de la solución

Para manejar las variables, los objetos y las funciones creamos tablas para cada una. Estas tablas se crean como una clase, que contienen distintas funciones para poder agregar a la tabla, buscar en la tabla, obtener un valor de la tabla y otras funciones independientes de cada clase.

La tabla de símbolos se encarga de almacenar las variables del código, todas las variables se almacenan con la misma estructura: variable, tipo, valor. Esta clase incluye funciones para agregar variables, asignar u obtener valores. Otras funciones que se pueden realizar en esta clase relacionadas con los objetos son para buscar un objeto en la tabla u obtener el valor de algún objeto. Todas estas funciones las explicaremos más adelante cuando se utilicen.

La tabla de registros almacena los objetos declarados en el código. Estos objetos se almacenan en formato tipo, valor. La clase incluye funciones para agregar registros a la tabla, buscar, y comprobar la estructura de los ajson. El funcionamiento de estas funciones se explicará más adelante.

La tabla de funciones se utiliza para almacenar las definiciones de las funciones que se hacen a lo largo del código. Se almacena el nombre de la función, el tipo de sus argumentos y el tipo de retorno.

Se ha hecho un tratamiento de errores específico que da información sobre la línea del error y el tipo del error. Aunque haya errores se puede seguir analizando el código que sea correcto. Se hace diferencia entre los errores léxicos, semánticos y sintácticos.

Análisis Semántico

Vamos a explicar con más detalle consideraciones del análisis semántico que hemos tenido en cuenta además de los errores que hemos contemplado.

Tipos simples

En el caso de los tipos simples como los números, booleanos o cadenas, siempre se almacenan como una lista que contiene valor en la primera posición y tipo en la segunda. De esta forma siempre que queramos acceder a un valor o un tipo en la tabla de símbolos, estos siempre estarán en el mismo orden.

En el caso de las variables, si la variable no se encuentra almacenada en el diccionario de locales, se buscará en la tabla de símbolos con la función obtener para conseguir su valor y su tipo. Si no se encuentra en esta tabla, se devolverá un error indicando que la variable no ha sido definida. Si la variable se encuentra en el diccionario de locales, significa que pertenece a una función y eso lo explicaremos más adelante.

Objetos ajson

Para los ajson, cuando estos se definen, se agrega a la tabla de registros el tipo del ajson y su valor. Esto se hace llamando a la función agregar_registro. Si se intenta guardar un ajson que ya ha sido definido se devolverá un error. Esta función también comprueba si los tipos de las claves del ajson definido son correctos. Por ejemplo, no puedes definir alguno de los tipos como un objeto que no ha sido declarado. En estos casos también devolverá un error indicando que los tipos son incorrectos.

En los ajson también se comprueban las claves introducidas, si se crea un ajson con las mismas claves aparecerá un error semántico. Tampoco se acepta un ajson vacío.

Notación punto y notación corchete

Estas notaciones sirven tanto para acceder al valor de un ajson como para modificarlo. Para implementar estas funciones lo primero que hacemos es que vamos almacenando todas las claves que se van identificando separadas por puntos. Por ejemplo:

- *var1.a[“prop1].l*, se almacena como “var1.a.prop1.l”. Lo hemos hecho así porque es una forma fácil de poder buscar cada clave en los ajson.

Las dos formas en las que se puede usar esta notación son:

- **Acceder a un valor:** en este caso lo que queremos es que se devuelva el valor del ajson en las claves especificadas. Para ello llamamos a una función recursiva de la tabla de símbolos, que es *obtener_valor_objeto*. A esta función le mandamos la cadena de claves separadas por puntos. En la primera iteración la función busca la primera clave en la tabla de símbolos. Si no se encuentra se devuelve un error ya que la variable no estaría declarada. En las siguientes iteraciones se van buscando el resto de claves en el ajson guardado en la tabla de símbolos. Si no

se encuentra alguna clave es que el ajson guardado no tiene esa estructura y se devuelve un error. En caso de que no haya ningún error se devolverá el valor y el tipo correcto.

- **Modificar un valor**: en este caso se llama a la función *buscar_objeto* y, además de la cadena de claves, se manda como argumento el tipo y valor de la variable. La primera parte de la función es igual que la anterior, busca el valor especificado por las claves y si no lo encuentra devuelve un error. En caso de que no haya error, se comprueba si el valor que se quiere cambiar tiene el mismo tipo que el que se quiere introducir. En caso contrario se devolverá un error. Si el tipo es el mismo se modificará el valor en la tabla de símbolos.

Operaciones y paréntesis

Las operaciones están divididas en aritméticas, binarias y de comparación.

Aritméticas

En el caso de las operaciones aritméticas, si se intenta hacer cualquier operación con valores que no tienen tipo o que no tienen valor, se mostrará el error correspondiente en cada caso.

Los tipos con los que se puede operar varían en función del tipo de operador, por ello hemos distinguido los casos en los que se opera con cada operador.

Para la **suma** y la **resta**, los tipos con los que se puede operar son los mismos. En primer lugar, comprobamos si alguno de los tipos en la operación es un float, ya que independientemente del otro tipo de dato con el que operemos, ya sea caracter o entero, el resultado va a ser de tipo float. En el enunciado se especifica que los caracteres se pueden transformar en enteros, y como los enteros se pueden transformar en floats, hemos considerado que la operación con caracteres y floats es correcta.

En el caso de que uno de los dos tipos sea entero ocurre lo mismo, pero primero comprueba si es float, por lo que en el caso en el que haya un entero y un decimal, el resultado será decimal.

Siempre que se opera con caracteres se realiza la operación con su correspondiente valor en ascii, de esta forma siempre se puede realizar la suma o la resta. En el caso en el que se sumen dos caracteres, como el código ascii solo va hasta el número 255, si se suman dos valores cuya suma es superior a 255, se realiza el módulo. Esto lo hemos decidido así ya que pensábamos que simplemente devolver el último caracter no aporta demasiada información.

En el caso de la **multiplicación**, no se permite la multiplicación de dos caracteres, por lo que si los dos operandos son caracteres se devolverá el error correspondiente. Al igual que en la suma y en la resta, primero se comprueba que uno de los dos valores sea de tipo float, si es así el resultado será de tipo float independientemente del otro valor. Esto también ocurre con los enteros. Si se trata de multiplicar un entero o un float por un caracter, como hemos explicado antes, se operará con el valor ascii del caracter.

Para las **divisiones** ocurre lo mismo que en las multiplicaciones, ya que no se permite una división entre caracteres. Tampoco se permite dividir entre 0. En ambos casos se muestran los errores correspondientes.

Binarias

En el caso de las operaciones **binarias**, al igual que en las aritméticas, las dividimos en tres casos distintos. En todos primero se comprueba si los tipos pasados a la operación son correctos, ya que los tipos deben ser obligatoriamente booleanos. En caso de que alguno de los dos tipos no sea booleano, se devolverá un error mostrando que los tipos no son compatibles. Si se trata de operar con variables que no tienen valor asignado, también se devolverá el error correspondiente. En caso de que los tipos sean los correctos y también tengan valor, se realizará la operación correspondiente, ya sea &&, || o !. En todos los casos se almacenará el resultado con el tipo booleano y el valor correspondiente a la operación, que puede ser fl o tr.

Comparación

Las operaciones de **comparación** se dividen en los operadores <, >, <=, >= o ==. En todos los casos, el resultado será de tipo booleano con valor tr o fl. Pueden aceptar comparaciones de distintos tipos. En el caso del operador ==, este acepta que los valores sean de tipo caracter, entero, float o boolean, pero no permite que se utilicen los de tipo boolean con el resto. Esto quiere decir que si alguno de los dos valores es de tipo boolean pero el otro no lo es, se mostrará el error de tipos incompatibles. Al igual que en las operaciones aritméticas, si se trata de comparar un caracter con cualquier otro tipo compatible, se convertirá a su valor en ascii. El resto de operaciones de comparación no aceptan utilizar valores de tipo boolean. El resto de la operación se realiza igual, convirtiendo a ascii los caracteres y comparando con el otro valor con tipo compatible.

En el caso en el que se realizan operaciones entre **paréntesis**, se le asigna más precedencia al valor entre paréntesis que a los valores que están fuera. Para los **signos**, a diferencia de la anterior entrega, estos se pueden situar delante de valores con tipos

enteros o float y se pueden poner tantos como se quiera, siempre van a cumplir las propiedades de los signos. Si se coloca un signo antes de cualquier otro tipo de dato se mostrará un error indicando que el tipo no es compatible. Esto lo hemos decidido así ya que creemos que no tiene sentido poder poner un signo delante de un caracter o un booleano.

Declaración

A la hora de declarar variables hacemos una diferencia entre declarar variables simples y objetos de tipo ajson.

Variables simples

Las variables se pueden declarar sin valor o asignarles un valor en la declaración. Para cualquiera de los casos, hay que comprobar que la variable no esté ya en la tabla de símbolos. En caso contrario se devolverá un error.

Si no se asigna un valor ni un tipo (*let a;*), la variable se almacenará en la tabla con tipo y valor None, que más adelante se podrá cambiar si se le asigna un valor. Cabe destacar que si no se le asigna un valor no se podrá usar en operaciones o en condicionales y bucles.

Si se asigna un tipo a una variable, esta se almacena en la tabla de símbolos con ese tipo y valor None. El tipo de una variable puede cambiar a lo largo del programa.

Si se asigna un valor a una variable simple cuando se declara (*let a = expr;*), se almacenará en la tabla de símbolos con el valor y tipo de la expresión. Se puede asignar cualquier expresión correcta a una variable, aunque hacemos una diferencia con los objetos ajson. Si se declara una variable con un tipo distinto al de la expresión (*let a:float = 9*) se almacena el tipo de la expresión.

Objetos ajson

Cuando declaramos un ajson, se le asigne o no un valor, siempre hay que especificar su tipo. Si se especifica un tipo de ajson tiene que estar definido previamente, si no se devolverá un error. Si no se asigna ningún valor se almacenará en la tabla con el tipo indicado y valor None.

En caso de que sí se le asigne un valor, se hace de forma distinta a una variable simple. La expresión que se asigna es el valor del ajson, pero en este caso no hemos almacenado en expresión el tipo. Es decir, para cualquier expresión simple se almacena de esta forma: [9, ‘int]. Si la expresión es un ajson no se almacena el tipo en la regla de “expr”, por lo que la longitud de la lista es 1 en vez de dos.

Esto nos permite hacer varias cosas. En primer lugar podemos diferenciar cuando una expresión es un ajson del resto, ya que las tenemos que tratar de forma distinta.

En las declaraciones del ajson sí que tenemos el tipo, ya que hemos dicho que es obligatorio especificarlo. Por lo tanto, en las declaraciones, si la longitud de la expresión es 1 estamos tratando con un ajson y llamamos a la función comprobar estructura de la tabla de registros. A esta función le pasamos como argumentos el ajson y el tipo con el que se ha declarado. Si la estructura coincide con la guardada en la tabla se almacenará el ajson con ese tipo en la tabla de símbolos. En caso contrario se devuelve un error.

En cuanto a los ajson anidados, hay dos formas de asignarles valores. Lo vamos a explicar con el siguiente ejemplo:

```
type objeto2 = {prueba: int};
type NestedObject = { "this is": int, prop2: int, "prop3": objeto2} ;
```

En este caso el segundo ajson tiene un valor que hace referencia al primer ajson. Se puede definir ese valor como una variable de ese objeto de esta forma:

```
let o: objeto2 = {prueba: 0};
let nested : NestedObject = {"this is": 30, prop2: 20, "prop3": o};
```

La otra forma de hacer es creando directamente la estructura en el segundo ajson:

```
let var2: NestedObject = { "this is": 10, prop2: 20, "prop3": {prueba: 10}};
```

Mientras el valor de “prop3” tenga la estructura del “objeto2” se acepta la declaración.

Asignación

En el caso de asignar valores a variables se hace de una forma distinta. No se puede asignar una variable si no está definida en la tabla de símbolos, a menos que sea el argumento de una función. Esto lo explicaremos más adelante, pero es por lo que se comprueba si la variable está en el diccionario de “locales”.

En cuanto a las variables simples, si ya están en la tabla de símbolos se puede cambiar tanto su valor como su tipo.

Los casos de las notaciones punto y corchete los explicaremos más adelante.

En el caso de los objetos ajson, se pueden reasignar si la estructura coincide, tanto las claves como los tipos de los valores. Por ello, al igual que en el apartado anterior, cuando la expresión es de longitud 1 se comprueba que la estructura coincide con la

que hay guardada en la tabla de registros. En este caso, como el ajson tiene que estar ya declarado, para obtener su tipo hay que sacarlo de la tabla de símbolos.

Condicionales y bucles

Respecto a los condicionales, se comprueba si existe una condición y si es de tipo booleano, en cualquiera de los dos casos se devolverá su error correspondiente. Las condiciones pueden ser variables o llamadas a funciones, siempre y cuando el valor devuelto sea de tipo boolean. No hemos implementado el control de flujo en el código, es decir, no se diferencia cuando se hace un if else, se realizan los dos caminos.

Para los bucles, ocurre lo mismo que en los condicionales, si no se especifica una condición para el bucle o si la condición especificada no es de tipo boolean, se mostrarán los errores correspondientes. Al igual que en los condicionales, también se acepta como condición una llamada a una función siempre que devuelva el tipo boolean. Como tampoco está programado, el código incluido en el statement de los bucles se realizará independientemente de la condición.

Definición de funciones

Cuando se declara una función correctamente se guarda en la tabla de funciones. Las funciones pueden o no tener argumentos pero siempre tienen que tener por lo menos la sentencia de retorno. Para considerar que la declaración de la función es correcta se comprueban estas condiciones:

- El tipo de retorno de la expresión tiene que ser el mismo que el que se define en la función. Si se devuelve alguno de los tipos simples solo hay que comprobar que el tipo de la función es el mismo que el de esa expresión.
- Para el caso de que se devuelva un objeto ajson hemos hecho algunas consideraciones adicionales. Un ajson se puede devolver de dos formas distintas. La primera es devolviendo una variable que ya estaba declarada, como en este ejemplo:

```
type Point = { x: int, y: int };
let punto: Point = { x:3, y:4 };
function get_point_after_jump(): Point {
    return punto;
}
```

  En este caso la variable “punto” ya está almacenada como un ajson de tipo “Point” en la tabla de símbolos y la expresión de retorno ya tiene ese tipo, por lo

que solo hay que compararlo con el tipo que devuelve la función (igual que con un tipo simple).

La otra forma en la que se acepta es esta:

```
type Point = { x: int, y: int };
function get_point_after_jump(a:int): Point {
    return { x:3, y:4 };
}
```

En este caso no se está devolviendo una variable ya definida. Por ello, lo que hacemos es que se compruebe si el ajson que se devuelve tiene la misma estructura que el que está almacenado en la tabla de registros. Para ello se llama a la función "comprobar_estructura" de la tabla de registros, pasando como argumentos la expresión de retorno y el tipo que devuelve la función. En este caso la expresión es correcta y es de tipo punto, por lo que se aceptaría. En caso de que no tuviera la misma estructura, o bien porque las claves estuvieran mal o bien porque los tipos no coinciden, se devolvería un error y la función no se guardaría en la tabla.

Si la expresión de retorno tiene cualquier otro tipo de error, como operaciones con tipos incorrectos, también se devuelve un error y no se almacena en la tabla.

- Todas las sentencias dentro de la función tienen que ser correctas o no se almacenará en la tabla.
- Después de comprobar estas condiciones se llama a la función "agregar" de la tabla de funciones. No se puede declarar una función con el mismo nombre y mismos argumentos que otra ya declarada. Por ello, cuando se llama a esta función se comprueba. Sí que se puede declarar dos funciones con el mismo nombre, pero tienen que tener distintos argumentos.

Si se cumplen todas estas condiciones se añade la función a la tabla de funciones. Lo que se guarda en la tabla es el nombre de la función, la lista de argumentos con sus tipos y el tipo de retorno. No guardamos los nombres de cada argumento porque no lo hemos considerado necesario. Cuando se llame a una función solo hay que comprobar los tipos de los argumentos, los nombres no son importantes.

En cuanto a los argumentos de las funciones, para permitir que se puedan usar en las funciones, hemos hecho lo siguiente. Tenemos un diccionario en el que cada vez que se inicia una función guardamos los argumentos, su nombre y su tipo. Para explicar el funcionamiento usaremos este ejemplo:

```
function gte(a: int, b: int): boolean {

    let c=a > b;
     return a > b;
}
```

En este caso, las variables a y b no tienen que estar en la tabla de símbolos, se pasan como argumentos de la función, por lo que se pueden usar dentro de la función sin estar declaradas. Es más, si las variables estuvieran declaradas con otro valor o tipo, no se tendrían que tener en cuenta ni usar ese valor. Las variables 'a' y 'b' en esta función son siempre de tipo entero.

Por ello, cuando se definen los argumentos de las funciones se guardan en el diccionario de "locales". Cada vez que se hace una asignación o una declaración en la que se utilice como expresión una variable, se comprueba primero si está en este diccionario. Si es así se usará esa variable y no se comprobará si está en la tabla de símbolos. Así conseguiremos que se puedan usar aunque no estén declaradas y que si están declaradas con otro tipo no se tengan en cuenta. Cuando se acaba una función se vacía el diccionario de locales porque esos argumentos solo son válidos para la función. Por lo tanto, si no estamos dentro de una función, el diccionario de locales estará vacío y se harán las comprobaciones de la tabla de símbolos.

No hemos implementado que los valores de los argumentos de las funciones sean de verdad los que se les pasa como argumento al llamarlas. Por lo tanto, hemos hecho que los valores de los argumentos sean siempre 0 si es un entero, 0.0 si es un float, 'a' si es un caracter y 'tr' si es un booleano. Es decir, en el ejemplo anterior, el valor de 'c' será 'fl' ya que a y b son 0.

En el caso de que se mande un ajson como argumento se puede acceder a sus valores con la notación punto y notación corchete. Sin embargo, si es un argumento no estará en la tabla de símbolos. En el diccionario de locales se almacenará el nombre del argumento y el tipo de ajson. Por lo tanto, para acceder a sus valores accedemos a la estructura guardada en la tabla de registros a través de la función obtener_valor_objeto. Esta función hace lo mismo que lo explicado en el apartado de notación punto y corchete, pero busca en la tabla de registros en vez de la de símbolos. Al igual que antes, si las claves no coinciden con la definición del ajson se devuelve un error. En caso contrario se devuelve el tipo del dato y se almacena el valor predeterminado para ese tipo.

Por último, no guardamos las variables que se pasan como argumentos en la tabla de símbolos porque son solo variables locales de la función, no se están declarando de forma global en el programa. Sin embargo, si se define una variable con la palabra 'let' en la función sí que se guardará en la tabla de símbolos.

Llamada a funciones

Al hacer una llamada a una función primero se comprueba que esté definida, en caso contrario se devuelve un error. Si está definida, después se comprueba que los argumentos son correctos, en caso contrario se devuelve un error. No hemos

implementado la llamada a la función real, no se hace el código de la función. Lo que hacemos cuando se llama a una función es que se almacena el tipo de retorno de la función en la variable indicada pero el valor es None.

Pruebas

En esta parte de la práctica no hemos modificado el analizador léxico, por lo que reconoce las mismas sentencias que en la anterior práctica. Los tests del 1 al 4 son los mismos que en la anterior entrega y son tests del analizador léxico. Los nuevos de esta entrega son los siguientes:

test_file5.ajson

Este test contiene declaraciones y asignaciones simples correctas. Se añaden los símbolos con el valor adecuado a la tabla de símbolos. Se permite que se cambie el tipo de las variables, como es el caso de la variable c, que primero se le asigna un valor entero pero después se cambia a un booleano. En la tabla de símbolos se muestra el último valor y tipo de todas las variables. También se pueden asignar variables a otras variables y se guardan con el mismo valor y tipo. Se comprueba también que se puedan poner múltiples signos delante de valores enteros o float.

test_file6.ajson

Este test contiene declaraciones de variables y asignaciones incorrectas. Se comprueba que no se pueda hacer una asignación de una variable que no esté declarada y que no se puedan declarar dos variables con el mismo nombre. También se comprueban errores estructurales, propios del parser, como la falta del caracter “;” al final de una sentencia, la falta del caracter “,” para separar variables o tratar de asignar valores en medio de una expresión. Además, se comprueba que no se puedan añadir signos de “+” o “-” delante de variables que no sean de tipo entero o booleano. Tampoco se puede asignar a una variable una cadena de caracteres delimitada por comillas dobles.

Aunque sea un test con errores, las variables que están bien declaradas sí que se guardan en la tabla de símbolos, como es el caso de ‘a’, ‘b’, y ‘variable_correcta’.

test_file7.ajson

Este test contiene operaciones aritméticas correctas. Los tipos que aceptan cada operación son los que hemos definido anteriormente y en este test se encuentran las combinaciones correctas. Se puede observar que el tipo en la tabla de símbolos después de cada operación es el explicado anteriormente. Además también comprobamos la precedencia de operadores. Por ejemplo, la asignación de la variable

'p' no tiene paréntesis, por lo que se hace primero la multiplicación y después la suma. En el caso de la variable 'q', se hacen primero las operaciones entre paréntesis.

test_file8.ajson

Este test contiene ejemplos de operaciones aritméticas incorrectas. Se comprueba que no se pueden multiplicar o dividir caracteres y que no se puede dividir entre 0. También se comprueba que no se pueden utilizar variables que no están definidas. Además, se prueba un ejemplo en el que se define una variable pero no se le asigna ni un tipo ni un valor, por lo que si se usa en una operación también se produce un error. Cuando se define una variable indicando su tipo, su valor inicial es None, por lo que si no se le asigna un valor tampoco se podrá utilizar en una operación.

test_file9.ajson

Este test contiene ejemplos de comparaciones correctas. Los tipos que se pueden utilizar en las comparaciones son los que se han explicado anteriormente y en este fichero se prueban las distintas combinaciones sin obtener errores. Al ser comparaciones, los resultados que se guardan en la tabla de símbolos son siempre booleanos y se puede comprobar que se asignan valores de verdadero o falso correctamente.

test_file10.ajson

Este test contiene comparaciones con errores. Se comprueban los errores en tipos no compatibles, como comparar un booleano con enteros, caracteres o float. Además, se comprueba que el signo "==" sólo acepte valores de tipo booleano. También se comprueba que no se aceptan en las comparaciones variables con valores nulos o con tipos nulos, es decir, que no se les ha asignado ningún valor.

test_file11.ajson

La primera parte de este test contiene expresiones binarias correctas. Los únicos tipos que admiten estas expresiones son booleanos y el resultado es siempre un booleano.

En la segunda parte del test se incluyen operaciones que mezcla los tres tipo de operaciones, pero siguiente la estructura y los tipos compatibles de cada operación. Por ejemplo, en esta operación: *let g = (3*9>9*9) || (2<3);* primero se realizan las operaciones entre paréntesis. Además, dentro de los primeros paréntesis se realizan antes las operaciones aritméticas y después las comparaciones. Finalmente se realiza la operación binaria de forma correcta ya que ambos operandos son booleanos. Si se quitan los paréntesis de la operación se hace en el mismo orden ya que es el que hemos definido. En el último ejemplo de este test hay varias operaciones binarias seguidas sin paréntesis, por lo que se evalúan de izquierda a derecha.

test_file12.ajson

Este test contiene operaciones binarias con errores. Se incluyen errores en los tipos de las variables o errores de variables no definidas. Muchas de estas operaciones y de las operaciones de errores anteriores sí que se aceptaban la entrega anterior, ya que sintácticamente son correctas pero tienen errores semánticos.

test_file13.ajson y test_file14.ajson

Estos dos tests tienen muchos casos de objetos correctos. Todos los objetos definidos correctamente se guardan en la tabla de registros con su tipo y su valor, que es toda la estructura del ajson. Hay ejemplos de las dos formas válidas de anidar objetos que hemos explicado anteriormente.

A la hora de declarar un ajson, se puede declarar sin inicializarlo, pero siempre hay que especificar su tipo. Si no se llega a inicializar se guarda con valor None. Los valores de los ajson pueden ser otras variables siempre que sean del tipo correcto. Por ejemplo, hemos probado estas dos asignaciones:

let o: objeto2 = {prueba: 0};
let nested : NestedObject = {"this is": 30, prop2: 20, "prop3": o};

Los tipos “objeto2” y “NestedObject” están correctamente definidos antes de estas asignaciones. En este caso el valor de la clave “prop3” es una variable de tipo objeto ya definida. En la tabla de registros se guarda una ‘o’, que es el nombre de la variable. Si luego accedemos al valor de esta forma: *let e = nested.prop3.prueba*; el valor que se almacena en la tabla de símbolos para la variable e es 0, el valor correcto de esa propiedad. La otra forma de inicializar un objeto de tipo NestedObject es así:

let var2: NestedObject = { "this is": 10, prop2: 20, "prop3": {prueba: 10}};

En este caso el valor de la clave “prop3” no es un objeto previamente creado, pero sí tiene la misma estructura que el objeto de tipo “object2”, por lo que se acepta. Es decir, aunque no sea una variable se comprueba que la estructura es correcta y si es así se acepta. Si accedemos al valor de esa propiedad de esta forma: *d = var2.prop3.prueba;* también se guardará en la tabla de símbolos el valor correspondiente, en este caso 10.

Hemos probado también la notación punto y corchete, tanto para obtener como para modificar los valores del ajson. Solo se permite modificar valores si mantienen el mismo tipo.

También se comprueba la reasignación del objeto. Para hacer esto tienen que coincidir todas las claves y tipos de las variables.

Hay también un ejemplo de ajson, el tipo “Objeto”, que tiene otros dos ajson anidados. Hemos probado el acceder a las variables de todos los objetos y se obtienen los valores adecuados.

test_file15.ajson

En este test se prueban errores en los objetos. Cada error muestra un mensaje específico del error para que sea más fácil identificarlos porque hemos probado muchos casos. Hemos comprobado lo siguiente:

- Se produce un error al definir un ajson vacío.
- Los tipos de los valores de un ajson sólo pueden ser los tipos del lenguaje u otros objetos ya definido, sino se produce un error.
- Las claves de un ajson no pueden estar repetidas.
- No se puede declarar una variable como un objeto que no estuviera previamente definido. Tampoco se puede declarar un objeto sin especificar su tipo.
- Al declarar y asignar un objeto, si no tiene la estructura definida se devuelve un error. Se comprueba que coincidan las claves y los tipos de cada valor.
- Si se intenta reasignar un objeto con tipos incorrectos en algún valor se devuelve un error. También se devuelve un error al intentar reasignar el objeto, el nuevo valor no cumple con la estructura del objeto.
- Se devuelve un error cuando se intenta cambiar el valor de una clave por un tipo distinto (usando la notación punto o corchete).
- Si se intenta acceder mediante la notación punto o corchete a un valor que no existe se devuelve un error.
- Se comprueban errores en las claves. Las claves solo pueden ser cadenas de caracteres sin comillas o con comillas dobles.
- También se comprueban problemas del parser, como no separar los pares de clave-valor con comas o no utilizar bien los corchetes para abrir y cerrar la definición de un objeto.
- Si se declara un objeto pero no se inicializa su valor es None, por lo que no se puede usar la notación punto o corchete para modificar el valor de las claves ni para asignarlo a otra variable.

test_file16.ajson

En este test se comprueban ejemplos de bucles y condicionales correctos. En estos casos solo hemos comprobado que la expresión de la condición tanto del bucle como del condicional sea de tipo booleano. El resto de sentencias dentro de cada bloque se reconocen de la misma forma que en el resto. Hemos comprobado que las condiciones

de los bucles sean “tr” o “fl”, que sean variables booleanas previamente definidas o valores de un objeto al que accedemos con la notación punto o corchete.

test_file17.ajson

Este test contiene ejemplos de errores en las estructuras condicionales y bucales. Comprobamos que se devuelva un error si la condición del bucle es cualquier otro tipo que no sea booleano. También comprobamos que no se puede asignar como valor de una variable un condicional o un bucle. Por último comprobamos que errores propios del parser como el mal uso de los corchetes o los paréntesis. Si hay errores en las sentencias dentro de un bucle o un condicional también se detecta.

test_file18.ajson

En este test probamos distintas declaraciones y funciones correctas. La localidad de las variables funciona como se ha explicado anteriormente. Los tipos que devuelven las funciones coinciden con su definición. Se pueden declarar nuevas variables dentro de las funciones. Dentro de una función puede también haber condicionales, bucles o cualquier otro tipo de sentencia. Se puede llamar a una función dentro de otra. También hemos probado a definir dos funciones con el mismo nombre y con distintos argumentos, lo que es correcto y se guardan como dos funciones distintas. A la hora de llamar a una función se comprueba que el tipo de los argumentos es el indicado en la definición.

Las funciones también pueden devolver objetos de ajson como hemos explicado antes y en este tests se comprueban los distintos casos.

test_file19.ajson

En este tests se prueban definiciones y llamadas a funciones con errores. Cada error da un mensaje específico. Se prueban los siguientes casos:

- Se intenta llamar a una función que no estaba definida y se devuelve un error.
- Se interna definir una función con el mismo nombre y argumentos que otra ya definida, lo que también devuelve un error.
- Si se llama a una función con distintos argumentos o distintos tipos a los que están en la definición también se devuelve un error.
- Si la función devuelve un tipo distinto al de la definición se devuelve un error.
- Si se utilizan variables que no estaban definidas ni son argumentos dentro de la función se devuelve un error.
- Por último se comprueban problemas estructurales que detecta el analizador sintáctico como la falta de paréntesis o corchetes.

Conclusión

Hemos diseñado analizadores léxico, sintáctico y semántico que permiten analizar código AlmostJavaScript, que contiene expresiones aritmeticológicas, objetos, funciones y estructuras de control básicas. Ha sido una práctica larga y nos ha parecido bastante compleja, sobre todo porque hemos tenido que realizarla durante el periodo de exámenes. Además hemos tenido en cuenta muchos tipos de errores en los tests y hemos tenido que ir modificando y mejorando la solución muchas veces.