uc3m | Universidad Carlos III de Madrid

P2 - AJS

Procesadores del Lenguaje

Fecha 28/04/2024

Curso III

Cuatrimestre 2º

Año 2023-2024

Titulación

Grado en Ingeniería Informática

Profesor

Victor Granados Harinero – vigranad@inf.uc3m.es

Integrantes

Santiago Kiril Cenkov Stoyanov

100472051 – **100472051@alumnos.uc3m.es**

Adrián Ruiz Albertos

100363617 – **100363617@alumnos.uc3m.es**

Repositorio

https://github.com/SanKiril/acad-Lang_Proc

Contenidos

Descripción de la práctica....3
Analizador léxico....3
Números....3
Enteros....3
Reales....3
Cadenas de caracteres....4
Explícitas....4
Implícitas....4
Comentarios....4
Carácter ASCII....4
Operadores aritméticos, booleanos y de comparación....4
Palabras reservadas....5
Delimitadores....5
Analizador sintáctico....5
Gramática....5
Ejecución....9
Entrada....9
Salida....9
Pruebas....9

Descripción de la práctica

La segunda y última práctica se construyó sobre la primera y consistió en diseñar e implementar un analizador léxico, sintáctico y semántico de AJS (Almost JavaScript), un lenguaje de programación personalizado y simple basado en JavaScript. El analizador utiliza los módulos `lex` y `yacc` de la librería Python **PLY (Python Lex-Yacc)**.

Analizador léxico

El analizador léxico está implementado en el fichero `ajs_lexer.py` y emplea el módulo `lex` para el análisis léxico. Este reconocerá:

Números

Se utilizan dos *tokens* para cada número puesto que más adelante en el análisis semántico será importante distinguirlos.

Enteros

Nombre: **`INTEGER`**

Características:

- Engloba a números binarios, octales, hexadecimales y decimales.
- Se sigue ese orden en la expresión regular para evitar el caso en el que una entrada es reconocida como el número decimal 0 y una cadena de caracteres, por ejemplo: `0b11`. Al evaluarse primero la expresión de binario, octal y hexadecimal antes que la decimal se evita ese problema.
- La forma decimal evita tener más de un cero a la izquierda.

Reales

Nombre: **`REAL`**

Características:

- Engloba a números científicos y flotantes.
- Otra vez se sigue ese orden especificado para evitar el caso en el que una entrada es reconocida como un número flotante y una cadena de caracteres debido a la letra `e` o `E` de la notación científica y la precedencia en la evaluación de la expresión regular.
- Una parte entera, ya sea la de coma flotante como la de la notación científica en el cuerpo o el exponente no pueden tener más de un cero a la izquierda.
- Se dan como válidos los casos en los que la parte entera existe y no existe/existe la parte decimal o no existe parte entera y sí parte decimal.
- Para no complicar el análisis sintáctico excesivamente, se ha decidido incluir el símbolo + o - en la parte exponencial de un número científico. Otra solución hubiera sido aplicar la operación unaria sobre un número científico en su conjunto pero también sobre solo la parte exponencial en el análisis sintáctico.

Ambos tipos de números no tienen signo durante el análisis léxico. De esta forma se simplifican las expresiones regulares de los *tokens* número y se traslada la lógica al análisis semántico donde se aplicarán operadores unarios para dar signo a todos los números de una forma más sencilla.

Cadenas de caracteres

Se utiliza un *token* para cada uno puesto que se necesitan diferenciar para el análisis sintáctico:

Explícitas

Nombre: **`STRING_EXPLICIT`**

Implícitas

Nombre: **`STRING_IMPLICIT`**

Comentarios

Nombre: **`COMMENT`**

Características:

- Son ignorados por el analizador léxico.

Carácter ASCII

Nombre: **`CHAR`**

Características:

- Está delimitado por comillas simples y reconoce un único carácter ASCII, inclusive el carácter comilla simple al formar parte de la tabla ASCII.

Operadores aritméticos, booleanos y de comparación

Durante la fase de diseño se barajaron 4 posibilidades:

- Definir tres *tokens*, uno para operadores aritméticos, otro para booleanos y otro para comparación.
- Definir dos *tokens*, uno para operadores binarios y otro para los unarios.
- No definir ningún token y utilizar literales por cada operador individual.
- Definir tokens por cada operador individual.

La primera opción complica el análisis sintáctico porque obliga a añadirle una capa de semántica resultando en una construcción del árbol sintáctico que no refleja la verdadera estructura del lenguaje AJS.

La segunda opción refleja a la perfección la estructura del lenguaje y se abstrae de la semántica. Sin embargo, esta opción no es factible porque el conjunto de los operadores binarios y los unarios no es disjunto. Por ejemplo, el símbolo menos puede actuar como operador binario o unario y no es posible conocer eso en el analizador léxico.

El único inconveniente de la tercera opción es que no recoge la estructura léxica real de todos los operadores, la cual habría que recomponerla en el análisis sintáctico. Por ejemplo, el operador de igualdad se forma con dos símbolos iguales y no con la concatenación de dos literales del símbolo igual en el análisis sintáctico.

Por tanto nos decantamos por la cuarta opción. Los tokens utilizados son **`PLUS`** (+), **`MINUS`** (-), **`TIMES`** (*), **`DIVIDE`** (/), **`NOT`** (!), **`AND`** (&&), **`OR`** (||), **`LE`** (<=), **`LT`** (<), **`EQ`** (==), **`GE`** (>=), **`GT`** (>), **`ASSIGN`** (=). Es importante definir primero los *tokens* más complejos como `LE` antes que `LT` y `GE` que `GT` para el correcto análisis léxico.

Palabras reservadas

Se han implementado como *tokens* reservados los cuales se reconocen exclusivamente en letras minúsculas. A algunos de ellos se les ha asignado valor: **`TR`** (valor: *True*) , **`FL`** y **`NULL`** (valor: *None*) para evitar tener que hacerlo posteriormente en el análisis.

Delimitadores

Se han implementado como literales por simplicidad.

Analizador sintáctico

El analizador sintáctico está implementado en el fichero `ajs_parser.py` y emplea el módulo `yacc` para el análisis sintáctico. Este empleará:

Gramática

```
file ::=
      statement file |
      block file |
      λ
      #El fichero puede estar vacío o no.
      #El fichero se compone de una/s sentencia/s de código y/o bloque/s de código.
statement ::=
      declaration ';' |
      assignment ';' |
      definition ';' |
      basic_expression ';'
      #Las sentencias de código siempre acaban en punto y coma.
block ::=
      simple_block |
      function
```

#Los bloques de código nunca acaban en punto y coma.

simple_block ::=

```
if_conditional |
while_loop
```

block_body ::=

```
block_body_nonempty |
λ
```

block_body_nonempty ::=

```
statement block_body_nonempty |
simple_block block_body_nonempty |
statement |
simple_block
```

#El cuerpo de un bloque permite el anidamiento de más sentencias y más bloques a excepción de funciones.

#El cuerpo de un bloque no puede estar vacío.

declaration ::=

```
LET declaration_content
```

declaration_content ::=

```
item ',' declaration_content |
item
```

#Se permite la declaración de múltiples variables a la vez, algunas pueden estar tipadas con algún objeto.

item ::=

```
STRING_IMPLICIT ':' STRING_IMPLICIT |
STRING_IMPLICIT
```

#Si una variable está tipada, lo está con un objeto y no con un tipo básico.

assignment ::=

```
declaration ASSIGN expression |
STRING_IMPLICIT ASSIGN expression
```

#Sí se permite la declaración y asignación a la vez de múltiples variables, pero no la asignación de múltiples variables solamente.

expression ::=

```
basic_expression | object
```

definition ::=

```
TYPE STRING_IMPLICIT ASSIGN definition_object
```

definition_object ::=

'{' definition_object_content '}'

definition_object_content ::=

definition_object_item ',' definition_object_content |

definition_object_item |

λ

definition_object_item ::=

key ':' type

object ::=

'{' object_content '}'

object_content ::=

object_item ',' object_content |

object_item |

λ

object_item ::=

key ':' expression

key ::=

STRING_EXPLICIT |

STRING_IMPLICIT

type ::=

INT |

FLOAT |

CHARACTER |

BOOLEAN |

STRING_IMPLICIT

if_conditional ::=

IF '(' basic_expression ')' '{' block_body_nonempty '}' |

IF '(' basic_expression ')' '{' block_body_nonempty '}' ELSE '{' block_body_nonempty '}'

while_loop ::=

WHILE '(' basic_expression ')' '{' block_body_nonempty '}'

function ::=

FUNCTION STRING_IMPLICIT '(' argument_list ')' ':' type '{' block_body RETURN expression ';' '}'

```
argument_list ::=
    argument_list_nonempty |
    λ
    #Se evita que la lista quede con una coma al final.
argument_list_nonempty ::=
    STRING_IMPLICIT ':' type ',' argument_list_nonempty |
    STRING_IMPLICIT ':' type
basic_expression ::=
    INTEGER |
    REAL |
    CHAR |
    TR |
    FL |
    NULL |
    STRING_IMPLICIT |
    '(' basic_expression ')' |
    PLUS basic_expression |
    MINUS basic_expression |
    NOT basic_expression |
    basic_expression PLUS basic_expression |
    basic_expression MINUS basic_expression |
    basic_expression TIMES basic_expression |
    basic_expression DIVIDE basic_expression |
    basic_expression AND basic_expression |
    basic_expression OR basic_expression |
    basic_expression LE basic_expression |
    basic_expression LT basic_expression |
    basic_expression EQ basic_expression |
    basic_expression GE basic_expression |
    basic_expression GT basic_expression |
    function_call |
    object_call
function_call ::=
    STRING_IMPLICIT '(' function_call_list ')'
```

```
function_call_list ::=
    function_call_list_nonempty |
    λ
    #Se evita que la lista quede con una coma al final.
function_call_list_nonempty ::=
    expression ',' function_call_list_nonempty |
    expression
object_call ::=
    STRING_IMPLICIT object_attribute_list
object_attribute_list ::=
    '[' STRING_EXPLICIT ']' object_attribute_list |
    '.' STRING_IMPLICIT object_attribute_list |
    '[' STRING_EXPLICIT ']' |
    '.' STRING_IMPLICIT
```

Precedencia

De arriba a abajo en orden de precedencia:

1. **UPLUS**, **UMINUS**, **NOT**: asociatividad por la derecha.

   #Ejemplo: `a + -b` *=>* *#OK:* `a + (-b)` *#NOK:* `(a + -b)`

2. **TIMES**, **DIVIDE**: asociatividad por la izquierda.

   #Ejemplo: `a * b / c` *=>* *#OK:* `(a * b) / c` *#NOK:* `a * (b / c)`

3. **PLUS**, **MINUS**: asociatividad por la izquierda.

   #Ejemplo: `a + b - c` *=>* *#OK:* `(a + b) - c` *#NOK:* `a + (b - c)`

   #Al tener un orden de precedencia inferior al de la multiplicación y al de la división, se asegura la ejecución correcta de las expresiones matemáticas.

4. **LE**, **LT**, **EQ**, **GE**, **GT**: sin asociatividad.

   #Ejemplo: `a < b > c` *=>* *#NOK:* `(a < b) > c` , `a < (b > c)`

5. **AND**, **OR**: asociatividad por la izquierda.

   #Ejemplo: `a || b && c` *=>* *#OK:* `(a || b) && c` *#NOK:* `a || (b && c)`

6. **ASSIGN**: asociatividad por la derecha.

   #Ejemplo: `a = b + c` *=>* *#OK:* `a = (b + c)` *#NOK:* `(a = b) + c`

Ejecución

Entrada

`python3 ./main.py <path>.ajs -<mode>`

- `<path>`: *ruta a un fichero AJS*
- `<mode>`: *lex = analizador léxico || par = analizador léxico, sintáctico y semántico.*

Salida

La salida de tanto el analizador léxico como léxico, sintáctico y semántico se halla contenida en un fichero de nombre igual que el de entrada y situado en el directorio `./output/`.

Pruebas

Los ficheros de prueba se encuentran en el directorio `./tests/`. A continuación se comentan los tests más interesantes:

- `test_ok_tokens.ajs`: prueba todas las formas correctas de los *tokens*. Cabe destacar los siguientes casos:
  - Los números reales admiten dos formas de escritura: con parte entera y con/sin parte decimal o sin parte entera y con parte decimal.
  - Tanto números enteros como reales no admiten varios ceros a la izquierda de una parte entera.
  - Los comentarios de una línea y de múltiple línea son ignorados completamente por el analizador léxico y por tanto no se muestran en el fichero de salida.
  - La variedad de entradas entre dobles comillas y simples comillas que son reconocidas como **`STRING_EXPLICIT`** y **`CHAR`** respectivamente y no como otros *tokens*.
  - A diferencia del **`STRING_EXPLICIT`**, el **`CHAR`** puede mostrar el carácter simple comilla que lo delimita puesto que es parte de los caracteres ASCII.
  - Las palabras reservadas escritas conteniendo alguna letra mayúsculas son reconocidas como **`STRING_IMPLICIT`** y no como una palabra reservada.
- `test_ok_comment_empty_file.ajs`: prueba que un fichero sea vacío a pesar de tener comentarios.
- `test_ok_expression.ajs`: prueba de forma extensa media un *test* generado

de forma automática que todas las expresiones sintácticas permitidas por el lenguaje lo son. Este *test* es la base de otros *tests*.