uc3m | Universidad Carlos III de Madrid

P2 - AJS (II)

Procesadores del Lenguaje

Fecha 30/05/2024

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

Introducción 3

Gramática 3

Precedencia 7

Implementación 7

Pruebas 9

Consideraciones adicionales 9

Conclusiones 9

Introducción

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)**.

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 ';' |
      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 assignment_content |
    STRING_IMPLICIT ASSIGN assignment_content |
    object_call ASSIGN assignment_content
    #Sí se permite la declaración y asignación a la vez de múltiples variables, pero no la asignación de múltiples variables solamente.
assignment_content ::=
    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 |
    definition_object_item ','
    #Se permite una coma al final.
definition_object_item ::=
    key ':' type
```

object ::=

'{' object_content '}'

object_content ::=

object_item ',' object_content |

object_item |

object_item ','

#Se permite una coma al final.

object_item ::=

key ':' assignment_content

key ::=

STRING_EXPLICIT |

STRING_IMPLICIT

type ::=

INT |

FLOAT |

CHARACTER |

BOOLEAN |

STRING_IMPLICIT

if_conditional ::=

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

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

while_loop ::=

WHILE '(' expression ')' '{' block_body_nonempty '}'

function ::=

FUNCTION function_head '{' block_body RETURN expression ';' '}'

function_head ::=

STRING_IMPLICIT '(' argument_list ')' ':' type

#Esta regla de producción permite definir funciones recursivas, puesto que para que sea correcta semánticamente esta estructura, es necesario que la función esté definida en el momento en el que se evalúa `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
expression ::=
    INTEGER |
    REAL |
    CHAR |
    TR |
    FL |
    NULL |
    STRING_IMPLICIT |
    '(' expression ')' |
    PLUS expression |
    MINUS expression |
    NOT expression |
    expression PLUS expression |
    expression MINUS expression |
    expression TIMES expression |
    expression DIVIDE expression |
    expression AND expression |
    expression OR expression |
    expression LE expression |
    expression LT expression |
    expression EQ expression |
    expression GE expression |
    expression GT 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`

Implementación

Se han implementado las clases `AJSObject` y `AJSOperator` para facilitar el análisis

semántico. `AJSObject` encapsula el tipo y valor de un objeto dentro del lenguaje AJS (para tipos básicos, tipos definidos y funciones). `AJSOperator` hereda de `AJSObject` e implementa la lógica de las operaciones entre objetos AJS.

A continuación se detallan las consideraciones realizadas en la implementación:

- **Expresiones:**
  - Se utiliza una instancia de `AJSOperator` entre dos `AJSObject` (expresión binaria) o solo a uno (expresión unaria).
  - Internamente `AJSOperator` recoge la compatibilidad del tipo de operador con los operandos y realiza una conversión automática de tipos de objetos AJS cuando esté permitido. La compatibilidad de los operadores y las conversiones implícitas que se realizan son los mismos que los especificados por el enunciado de la práctica.
  - Las operaciones se resuelven dando un nuevo objeto AJS donde el tipo siempre será conocido pero el valor puede calcularse o no dependiendo de si existe algún operando que tenga o no valor.
- **Asignaciones:**
  - Cuando se realizan asignaciones al valor nulo, los objetos AJS pierden su tipo excepto si han sido declarados como un tipo definido.
  - Antes de asignar objetos definidos se comprueba que la estructura del objeto cumple con la especificada por el tipo de la variable a la que se está asignando. Se comprueba que el número de atributos sea el mismo, el nombre de todos ellos sea el mismo y que el valor de estos sea el definido. Esta comprobación se realiza de forma recursiva puesto que se puede definir como tipo de un atributo otro objeto definido. Sí está permitido alterar el orden de los atributos en la asignación respecto al de la definición.
  - Se permite multi declaraciones y asignaciones de objetos siempre y cuando la estructura del literal objeto al que se asigna cumpla con la de todas las variables declaradas.
- **Funciones:**
  - Las llamadas a funciones devuelven un `AJSOperator` con tipo igual al tipo de retorno de la función y valor nulo puesto que su cuerpo no se ejecuta.

- Durante el análisis semántico de una función se utilizan como variables los argumentos de la función sobre variables con mismo nombre declaradas en el ámbito global. Tras su ejecución se vuelve a usar las variables del contexto global.

Pruebas

Los ficheros de prueba se encuentran en los subdirectorios del directorio `./tests/`. Este tiene como subdirectorios: `./lexical/`, *tests* para el analizador léxico; `./syntactic/`, *tests* para el analizador léxico y sintáctico; y `./semantic/`, *tests* para el analizador léxico, sintáctico y semántico. Los *tests* dentro de `./lexical/` y `./semantic/` se reutilizan de la primera parte de la práctica. Todos ellos contienen explicaciones de qué se está probando.

Consideraciones adicionales

No se ha implementado el ámbito de los distintos bloques del lenguaje AJS (bloque de definición de función, bloque condicional y bloque de bucle). Esta parte quedaría como trabajo futuro y permitiría que las sentencias dentro de los distintos bloques fueran evaluadas en el momento adecuado (la definición, declaración o asignación dentro de un bloque sólo se tendría que dar en tiempo de ejecución).

Conclusiones

La realización de la práctica final nos ha permitido comprender la complejidad que existe tras la construcción de un compilador. Existen multitud de escenarios a tener en cuenta y su correcto funcionamiento es crítico pues de él dependen todos los programas escritos para el lenguaje del compilador.

Esa complejidad se ha visto en parte reducida por las herramientas usadas y la relativa simplicidad del lenguaje AJS. Utilizando Python para la implementación evita gestionar la memoria del lenguaje. Sin embargo, limita las posibilidades del compilador, por ejemplo, para la implementación de funciones o de clases. También no permite implementar optimizaciones avanzadas durante la compilación como *loop unrolling*, *out-of-order execution*, *data prefetching*, etc.

Respecto al tiempo invertido, se estima que han sido necesarias 30 horas de diseño, codificación, depuración, validación y documentación para la totalidad de la práctica final.