Lenguajes de Programación

Generación de código ejecutable

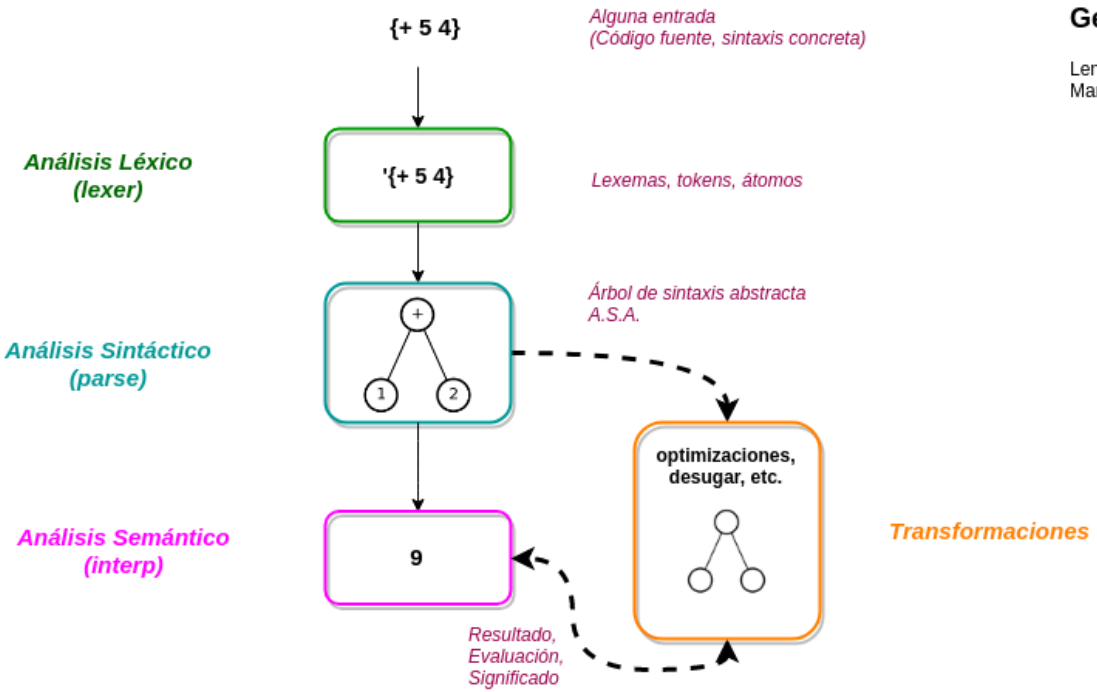
Análisis léxico y sintáctico

Manuel Soto Romero

Universidad Nacional Autónoma de México
Facultad de Ciencias

12 de septiembre de 2018

Generación de código ejecutable



Generación de Código Ejecutable

Lenguajes de Programación
Manuel Soto Romero

Análisis Léxico

Se convierte una secuencia de caracteres en una secuencia de lexemas.

Análisis Sintáctico

Se analizan los lexemas de acuerdo a las reglas de la gramática y se construye el Árbol de Sintaxis Abstracta (A.S.A.)

Análisis Sintáctico

El intérprete recorre el A.S.A. y calcula su valor aplicando los operadores correspondientes a los nodos hijos

Transformaciones

Dado el A.S.A. se pueden aplicar transformaciones para realizar optimizaciones o para quitar azúcar sintáctica, por ejemplo.

Sintaxis concreta

Se refiere a cómo se escriben expresiones en un lenguaje de programación dada una gramática

La Forma Extendida de Backus Naur es la forma más usada para describir la gramática de un lenguaje de programación.

```
<expr> ::= <id>
         | <num>
         | {<binop> <expr> <expr>}
         | {with {<id> <expr>} <expr>}

<id> ::= a | ... | z | A | ... | Z | aa | ...

<num> ::= ... | -2 | -1 | 0 | 1 | 2 | 3 | ...

<binop> ::= + | - | * | /
```

Análisis léxico

Es el primer tipo de análisis por el que pasa un código. Consiste en tomar la sintaxis concreta de un programa y separarla en lexemas (*átomos*).

El programa encargado de hacer esta separación es llamado *analizador léxico* (*scanner*, *lexer*).

Analizador léxico sencillo

Un analizador léxico sencillo regresa los siguientes lexemas para la expresión `{+ 17 29}`:

```
[{, +, 17, 29, }]
```

Analizador léxico avanzado

Un analizador léxico avanzado regresa los siguientes lexemas para la expresión `a = a + b`:

```
[(id,a),(asig,=),(id,a),(op,+),(id,b)]
```

Primitiva `quote`

Ejemplo

Para ahorrar el trabajo de obtener los lexemas de una expresión, en Racket, se puede hacer uso de la primitiva `quote`.

Tómalo literal, no lo interpretes

En la frase: ***El barco del ayudante***

¡Tómalo literal!

¡No lo interpretes!

Primitiva `quote`

Ejemplo (serio)

Sin quote

```
> {+ 17 29}
46
```

Con quote

```
> (quote {+ 17 29})
'{+ 17 29}
> '{+ 17 29}
'{+ 17 29}
```

```
'{   +    17    29   }
   first second third
```

`quote` regresa: números, símbolos o listas (de símbolos).

Primitiva `quote`

Entrada del usuario: read

Decíamos que el analizado léxico toma una cadena y la separa en lexemas, pero `quote` no toma cadenas.

Racket cuenta con una primitiva que pide una cadena al usuario (del teclado) y le aplica `quote`, llamada `read`.

```
> (read)
{+ 1 2}
'{+ 1 2}
```

Análisis sintáctico

La sintaxis concreta con que se definen las expresiones de un lenguaje es clara para el usuario, pero no mucho para la computadora.

```
1 + 2
+ 1 2
1 2 +
```

La abstracción es un principio por el cual se ignora toda aquella información que no es relevante. En este caso la posición del operador.

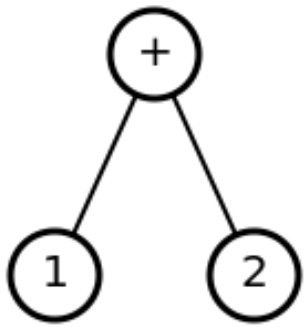
Se usan representaciones intermedias para abstraer este tipo de situaciones. En el curso de Lenguajes usaremos *Árboles de Sintaxis Abstracta* (ASA).

Análisis sintáctico

Ejemplo

Para las expresiones anteriores

```
1 + 2
+ 1 2
1 2 +
```




```
(add (num 1) (num 2))
```

ASA para WAE

Para representar los ASA de los lenguajes que se implementarán a lo largo del curso, se usará la primitiva `define-type`.

```
;; TDA para representar los árboles de sintaxis
;; abstracta del lenguaje WAE.
(define-type WAE
    [id     (id symbol?)]
    [num    (n numero?)]
    [binop (f operador?) (izq WAE?) (der WAE?)]
    [with   (id symbol?) (value WAE?) (body WAE?)])
```

Cada constructor representará la raíz del ASA y sus parámetros serán los hijos.

Analizador sintáctico para WAE

Este análisis consiste en tomar una expresión en sintaxis concreta y construir el árbol de sintaxis abstracta correspondiente (un mapeo).

```
;; Función que toma una expresión en sintaxis concreta
;; y regresa el árbol de sintaxis abstracta
;; correspondiente.
;; parse: s-expresion -> WAE
(define (parse sexp)
    (match sexp
        [(? symbol?) ...]
        [(? number?) ...]
        [(list 'with (list id value) body) ...]
        [(list f izq der) ...]))
```

Analizador sintáctico para WAE

Símbolos.

Un símbolo se mapea a un `id`.

```
;; Función que toma una expresión en sintaxis concreta
;; y regresa el árbol de sintaxis abstracta
;; correspondiente.
;; parse: s-expresion -> WAE
(define (parse sexp)
    (match sexp
        [(? symbol?) (id sexp)]
        [(? number?) ...]
        [(list 'with (list id value) body) ...]
        [(list f izq der) ...]))
```

Analizador sintáctico para WAE

Números.

Un número se mapea a un `num`.

```
;; Función que toma una expresión en sintaxis concreta
;; y regresa el árbol de sintaxis abstracta
;; correspondiente.
;; parse: s-expresion -> WAE
(define (parse sexp)
    (match sexp
        [(? symbol?) (id sexp)]
        [(? number?) (num sexp)]
        [(list 'with (list id value) body) ...]
        [(list f izq der) ...]))
```

Analizador sintáctico para WAE

Operaciones binarias.

Una operación binaria se mapea a un `binop`. Hay que mapear el símbolo que representa el operador en una función de Racket y recursivamente procesar el lado izquierdo y derecho.

```
;; Función que toma una expresión en sintaxis concreta
;; y regresa el árbol de sintaxis abstracta
;; correspondiente.
;; parse: s-expresion -> WAE
(define (parse sexp)
   (match sexp
      [(? symbol?) (id sexp)]
      [(? number?) (num sexp)]
      [(list 'with (list id value) body) ...]
      [(list f izq der)
         (binop (elige f) (parse izq) (parse der))]))
```

Analizador sintáctico para WAE

Asignaciones locales.

Una asignación local se mapea a un `with`. Hay que construir el árbol `with` con el identificador y procesar recursivamente el valor asociado al identificador y el cuerpo del `with`.

```
;; Función que toma una expresión en sintaxis concreta
;; y regresa el árbol de sintaxis abstracta
;; correspondiente.
;; parse: s-expresion -> WAE
(define (parse sexp)
   (match sexp
      [(? symbol?) (id sexp)]
      [(? number?) (num sexp)]
      [(list 'with (list id value) body)
         ¿?]
      [(list f izq der)
         (binop (elige f) (parse izq) (parse der))]))
```