Laboratorio de Lenguajes de Programación

Generando código ejecutable - Parte I

Manuel Soto Romero

Universidad Nacional Autónoma de México
Facultad de Ciencias

7 de septiembre de 2016

Objetivos

Objetivos

- Recordar definiciones relacionadas con el proceso de generación de código ejecutable.

Objetivos

- Recordar definiciones relacionadas con el proceso de generación de código ejecutable.
- Recordar en qué consiste el análisis léxico.

Objetivos

- Recordar definiciones relacionadas con el proceso de generación de código ejecutable.
- Recordar en qué consiste el análisis léxico.
- Recordar en qué consiste el análisis sintáctico.

Objetivos

- Recordar definiciones relacionadas con el proceso de generación de código ejecutable.
- Recordar en qué consiste el análisis léxico.
- Recordar en qué consiste el análisis sintáctico.
- Implementar un parser sencillo para la gramática `FWAE`.

¿Qué es un lenguaje de programación?

¿Qué es un lenguaje de programación?

Definition

Un lenguaje de programación es en pocas palabras, una interfaz entre los humanos y las computadoras.

¿Qué es un lenguaje de programación?

Definition

Un lenguaje de programación es en pocas palabras, una interfaz entre los humanos y las computadoras.

Un lenguaje de programación consiste de:

¿Qué es un lenguaje de programación?

Definition

Un lenguaje de programación es en pocas palabras, una interfaz entre los humanos y las computadoras.

Un lenguaje de programación consiste de:

- Sintaxis
- Semántica
- Convenciones de programación (*idioms*)
- Bibliotecas

¿Qué es un lenguaje de programación?

Definition

Un lenguaje de programación es en pocas palabras, una interfaz entre los humanos y las computadoras.

Un lenguaje de programación consiste de:

- Sintaxis
- Semántica
- Convenciones de programación (*idioms*)
- Bibliotecas

En este curso nos dedicamos a estudiar la *semántica* mediante el diseño e implementación de intérpretes

Tipos de análisis para generar código ejecutable

Tipos de análisis para generar código ejecutable

Para obtener código ejecutable dado un código fuente, necesitamos de distintos tipos de análisis:

Tipos de análisis para generar código ejecutable

Para obtener código ejecutable dado un código fuente, necesitamos de distintos tipos de análisis:

- Análisis léxico.

Tipos de análisis para generar código ejecutable

Para obtener código ejecutable dado un código fuente, necesitamos de distintos tipos de análisis:

- Análisis léxico.
- Análisis sintáctico.

Tipos de análisis para generar código ejecutable

Para obtener código ejecutable dado un código fuente, necesitamos de distintos tipos de análisis:

- Análisis léxico.
- Análisis sintáctico.
- Análisis semántico

Tipos de análisis para generar código ejecutable

Para obtener código ejecutable dado un código fuente, necesitamos de distintos tipos de análisis:

- Análisis léxico.
- Análisis sintáctico.
- Análisis semántico
- Optimizaciones

Tipos de análisis para generar código ejecutable

Para obtener código ejecutable dado un código fuente, necesitamos de distintos tipos de análisis:

- Análisis léxico.
- Análisis sintáctico.
- Análisis semántico
- Optimizaciones

En esta sesión nos dedicaremos a estudiar el análisis léxico y sintáctico

Análisis léxico

Análisis léxico

Dado un código fuente, el análisis léxico se encarga de separar el mismo en lexemas (*tokens*).

Análisis léxico

Dado un código fuente, el análisis léxico se encarga de separar el mismo en lexemas (*tokens*).

Por ejemplo, para el siguiente código en Java:

Análisis léxico

Dado un código fuente, el análisis léxico se encarga de separar el mismo en lexemas (*tokens*).

Por ejemplo, para el siguiente código en Java:

```
if (10 < 1729) {
   int a = 10;
}
```

Análisis léxico

Dado un código fuente, el análisis léxico se encarga de separar el mismo en lexemas (*tokens*).

Por ejemplo, para el siguiente código en Java:

```
if (10 < 1729) {
   int a = 10;
}
```

Un analizador sintáctico (Lexer o Scanner) regresaría:

Análisis léxico

Dado un código fuente, el análisis léxico se encarga de separar el mismo en lexemas (*tokens*).

Por ejemplo, para el siguiente código en Java:

```
if (10 < 1729) {
   int a = 10;
}
```

Un analizador sintáctico (Lexer o Scanner) regresaría:

```
[(palabra-reservada, if), (paréntesis-a, ( ), (entero, 10),
(operador-binario, <), (entero, 1729), (parentesis-c, ) ),
(llave-a, {), (palabra-reservada, int), (identificador, a),
(operador-binario, =), (pycoma, ;), (llave-c, {)]
```

Análisis léxico

Dado un código fuente, el análisis léxico se encarga de separar el mismo en lexemas (*tokens*).

Por ejemplo, para el siguiente código en Java:

```
if (10 < 1729) {
   int a = 10;
}
```

Un analizador sintáctico (Lexer o Scanner) regresaría:

```
[(palabra-reservada, if), (paréntesis-a, ( ), (entero, 10),
(operador-binario, <), (entero, 1729), (parentesis-c, ) ),
(llave-a, {), (palabra-reservada, int), (identificador, a),
(operador-binario, =), (pycoma, ;), (llave-c, {)]
```

La notación `quote` y la función `read`, nos ahorran este trabajo en Racket

read y quote

`read` y `quote`

Los lenguajes de programación funcionales que forman parte de la familia de Lisp, hacen uso de una primitiva llamada `quote`. Esta notación permite escribir expresiones complejas sin que se evalúen, lo que nos permite trabajar con ellas como si fueran símbolos o listas.

`read` y `quote`

Los lenguajes de programación funcionales que forman parte de la familia de Lisp, hacen uso de una primitiva llamada `quote`. Esta notación permite escribir expresiones complejas sin que se evalúen, lo que nos permite trabajar con ellas como si fueran símbolos o listas.

En pocas palabras, cuando usamos `quote` estamos diciendo:

`read` y `quote`

Los lenguajes de programación funcionales que forman parte de la familia de Lisp, hacen uso de una primitiva llamada `quote`. Esta notación permite escribir expresiones complejas sin que se evalúen, lo que nos permite trabajar con ellas como si fueran símbolos o listas.

En pocas palabras, cuando usamos `quote` estamos diciendo:

"Tómalo literal, no lo interpretes"

read y quote

Los lenguajes de programación funcionales que forman parte de la familia de Lisp, hacen uso de una primitiva llamada `quote`. Esta notación permite escribir expresiones complejas sin que se evalúen, lo que nos permite trabajar con ellas como si fueran símbolos o listas.

En pocas palabras, cuando usamos `quote` estamos diciendo:

"Tómalo literal, no lo interpretes"

Por ejemplo, en la frase:

read y quote

Los lenguajes de programación funcionales que forman parte de la familia de Lisp, hacen uso de una primitiva llamada `quote`. Esta notación permite escribir expresiones complejas sin que se evalúen, lo que nos permite trabajar con ellas como si fueran símbolos o listas.

En pocas palabras, cuando usamos `quote` estamos diciendo:

"Tómalo literal, no lo interpretes"

Por ejemplo, en la frase:

"El perro del ayudante"

read y quote

`read` y `quote`

Si usamos la notación `quote`:

`read` y `quote`

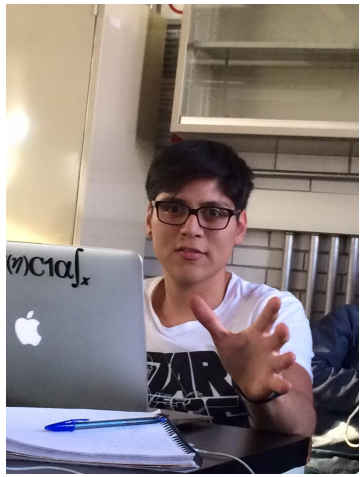
Si usamos la notación `quote`:

¡Tómalo literal!

read y quote

Si usamos la notación quote:

¡Tómalo literal!



¡No lo interpretes!

read y quote

`read` y `quote`

Ahora un ejemplo en serio...

read y quote

Ahora un ejemplo en serio...

La expresión `(+ (- 20 3) 8)` se evalúa a:

read y quote

Ahora un ejemplo en serio...

La expresión `(+ (- 20 3) 8)` se evalúa a:

25

read y quote

Ahora un ejemplo en serio...

La expresión `(+ (- 20 3) 8)` se evalúa a:

25

Si usamos `quote`:

read y quote

Ahora un ejemplo en serio...

La expresión `(+ (- 20 3) 8)` se evalúa a:

`25`

Si usamos `quote`:

`'(+ (- 20 3) 8)` o `(quote (+ (- 20 3) 8))`

read y quote

Ahora un ejemplo en serio...

La expresión `(+ (- 20 3) 8)` se evalúa a:

25

Si usamos `quote`:

`'(+ (- 20 3) 8)` o `(quote (+ (- 20 3) 8))`

`'(  +  (- 20 3)  8  )`

first **second** **third**

read y quote

Ahora un ejemplo en serio...

La expresión `(+ (- 20 3) 8)` se evalúa a:

25

Si usamos `quote`:

`'(+ (- 20 3) 8)` o `(quote (+ (- 20 3) 8))`

`'(  +  (- 20 3)  8  )`

first **second** **third**

La expresión no se evalúa y nos regresa una expresión en forma de lista, esto nos permite "tokenizar" las expresiones más fácil

read y quote

`read` y `quote`

La primitiva `read` aplica `quote` a las expresiones que ingresa el usuario desde el teclado, por ejemplo:

`read` y `quote`

La primitiva `read` aplica `quote` a las expresiones que ingresa el usuario desde el teclado, por ejemplo:

```
> (read)
(+ 2 3)
```

read y quote

La primitiva `read` aplica `quote` a las expresiones que ingresa el usuario desde el teclado, por ejemplo:

```
> (read)
(+ 2 3)

'(+ 2 3)
```

`read` y `quote`

La primitiva `read` aplica `quote` a las expresiones que ingresa el usuario desde el teclado, por ejemplo:

```
> (read)
(+ 2 3)

'(+ 2 3)
```

Usamos entonces estas primitivas para ahorrarnos el análisis léxico. Esto se estudia a profundidad en un curso de compiladores mediante el uso de autómatas

Análisis sintáctico

Análisis sintáctico

Un analizador sintáctico, a partir de las expresiones devueltas por el analizador lexico, convierte sintaxis concreta en abstracta.

Análisis sintáctico

Un analizador sintáctico, a partir de las expresiones devueltas por el analizador lexico, convierte sintaxis concreta en abstracta.

¿Cuál es la diferencia entre sintaxis concreta y sintaxis abstracta?

Análisis sintáctico

Un analizador sintáctico, a partir de las expresiones devueltas por el analizador lexico, convierte sintaxis concreta en abstracta.

¿Cuál es la diferencia entre sintaxis concreta y sintaxis abstracta?

Sintaxis concreta:

Análisis sintáctico

Un analizador sintáctico, a partir de las expresiones devueltas por el analizador lexico, convierte sintaxis concreta en abstracta.

¿Cuál es la diferencia entre sintaxis concreta y sintaxis abstracta?

Sintaxis concreta:

```
(+ 2 3)
4 + 7
4 5 +
```

Análisis sintáctico

Un analizador sintáctico, a partir de las expresiones devueltas por el analizador lexico, convierte sintaxis concreta en abstracta.

¿Cuál es la diferencia entre sintaxis concreta y sintaxis abstracta?

Sintaxis concreta:

```
(+ 2 3)
4 + 7
4 5 +
```

Sintaxis abstracta:

Análisis sintáctico

Un analizador sintáctico, a partir de las expresiones devueltas por el analizador lexico, convierte sintaxis concreta en abstracta.

¿Cuál es la diferencia entre sintaxis concreta y sintaxis abstracta?

Sintaxis concreta:

```
(+ 2 3)
4 + 7
4 5 +
```

Sintaxis abstracta:

```
(suma (numero 2) (numero 3))
(suma (numero 4) (numero 7))
(suma (numero 4) (numero 5))
```

El lenguaje FWAE en BNF

El lenguaje FWAE en BNF

Para ilustrar el proceso de generación de código ejecutable. Definamos las reglas de construcción de las expresiones del lenguaje FWAE (Function With Arithmetic Expresion) en BNF.

El lenguaje FWAE en BNF

Para ilustrar el proceso de generación de código ejecutable. Definamos las reglas de construcción de las expresiones del lenguaje FWAE (Function With Arithmetic Expresion) en BNF.

```
<FWAE> ::= <num>
         | {<bin-op> <FWAE> <FWAE>}
         | {with {<id> <FWAE>} <FWAE>}
         | <id>
         | {fun {<id>} <FWAE>}
         | {<FWAE> <FWAE>}
<num> ::= 0 | 1 | 2 | 3 | 4 | 5 ...
<bin-op> ::= + | * | - | / | % | max | min | pow
<id> ::= a | b | c | d | ...
```

Ejemplos de expresiones en FWAE

Ejemplos de expresiones en FWAE

```
(λ) 1
(λ) -7
(λ) {+ 2 9}
(λ) {- 1 7}
(λ) {* 2 9}
(λ) {/ {+ 1 7} {* 2 9}}
(λ) a
(λ) b
(λ) x
(λ) {% 10 2}
(λ) {pow 2 3}
(λ) {max 1729 1835}
(λ) {min 1729 1835}
(λ) {with {x 2}
       {* x x}}
(λ) {fun {x} {pow x 2}}
(λ) {{fun {x} {pow x}} 2}
```

Ejemplos de expresiones en FWAE

```
(λ) 1
(λ) -7
(λ) {+ 2 9}
(λ) {- 1 7}
(λ) {* 2 9}
(λ) {/ {+ 1 7} {* 2 9}}
(λ) a
(λ) b
(λ) x
(λ) {% 10 2}
(λ) {pow 2 3}
(λ) {max 1729 1835}
(λ) {min 1729 1835}
(λ) {with {x 2}
       {* x x}}
(λ) {fun {x} {pow x 2}}
(λ) {{fun {x} {pow x}} 2}
```

Para diferenciar el prompt de Racket del de FWAE usamos (λ) y usamos llaves en lugar de paréntesis.

El lenguaje FWAE en sintaxis abstracta

El lenguaje FWAE en sintaxis abstracta

```
(define-type FWAE
   [num (number?)]
   [bin-op (f procedure?)  (l FWAE?) (r FWAE?)]
   [with (name symbol?)  (named-expr FWAE?) (body FWAE?)]
   [id (name symbol?)]
   [fun (param symbol?)  (body FWAE?)]
   [app (fun-expr FWAE?) (arg-expr FWAE?)])
```

El lenguaje FWAE en sintaxis abstracta

```
(define-type FWAE
   [num (number?)]
   [bin-op (f procedure?)  (l FWAE?) (r FWAE?)]
   [with (name symbol?)  (named-expr FWAE?) (body FWAE?)]
   [id (name symbol?)]
   [fun (param symbol?)  (body FWAE?)]
   [app (fun-expr FWAE?) (arg-expr FWAE?)])
```

El objetivo es escribir un analizador sintáctico que convierta expresiones de la gramática en BFN a la sintaxis abstracta que acabamos de definir. Por ejemplo:

El lenguaje FWAE en sintaxis abstracta

```
(define-type FWAE
   [num (number?)]
   [bin-op (f procedure?)  (l FWAE?) (r FWAE?)]
   [with (name symbol?)  (named-expr FWAE?) (body FWAE?)]
   [id (name symbol?)]
   [fun (param symbol?)  (body FWAE?)]
   [app (fun-expr FWAE?) (arg-expr FWAE?)])
```

El objetivo es escribir un analizador sintáctico que convierta expresiones de la gramática en BFN a la sintaxis abstracta que acabamos de definir. Por ejemplo:

```
1 -> (num 1)
```

```
{max 1729 1835} -> (bin-op max (num 1729) (num 1835))
```

```
{fun {x} {pow x 2}} -> (fun 'x (bin-op expt 'x (num 2)))
```

Definiendo el analizador sintáctico

Definiendo el analizador sintáctico

```
(define (parse sexp)
   (cond
      [(symbol?  sexp) (id sexp)]
      [(number?  sexp) (num sexp)]
      [(list?  sexp)
         (case (car sexp)
            [(+ * - / % max min pow) (bin-op
                                      (elige (car sexp))
                                      (parse (first sexp))
                                      (parse (second sexp)))]
            [(with) ...]
            [(fun) ...]
            [else ...]))
```

La función `elige`

La función `elige`

```
(define (elige s)
   (case s
      [(+) +]
      [(-) -]
      [(*) *]
      [(/) /]
      [(%) modulo]
      [(max) max]
      [(min) min]
      [(pow) expt]))
```

Ejemplo

Ejemplo

```
(parse '{pow 2 3})
```

Ejemplo

```
(parse '{pow 2 3})
```

```
(symbol?  '{pow 2 3})
```
NO.

Ejemplo

```
(parse '{pow 2 3})
```

```
(symbol?  '{pow 2 3})
```
NO.

```
(number?  '{pow 2 3})
```
NO.

Ejemplo

`(parse '{pow 2 3})`

`(symbol? '{pow 2 3})` NO.

`(number? '{pow 2 3})` NO.

`(list? '{pow 2 3})` Sí. Entonces, entramos al `cond` con `(car sexp)` `= 'pow`

Ejemplo

`(parse '{pow 2 3})`

`(symbol? '{pow 2 3})` NO.

`(number? '{pow 2 3})` NO.

`(list? '{pow 2 3})` Sí. Entonces, entramos al `cond` con `(car sexp)` `= 'pow`

Caza con `(+ * - / % max min pow)`, entonces:

Ejemplo

`(parse '{pow 2 3})`

`(symbol? '{pow 2 3})` NO.

`(number? '{pow 2 3})` NO.

`(list? '{pow 2 3})` Sí. Entonces, entramos al `cond` con `(car sexp)` `= 'pow`

Caza con `(+ * - / % max min pow)`, entonces:

```
(bin-op (elige 'pow) (parse 2) (parse 3))
```

Ejemplo

`(parse '{pow 2 3})`

`(symbol? '{pow 2 3})` NO.

`(number? '{pow 2 3})` NO.

`(list? '{pow 2 3})` Sí. Entonces, entramos al `cond` con `(car sexp) = 'pow`

Caza con `(+ * - / % max min pow)`, entonces:

```
(bin-op (elige 'pow) (parse 2) (parse 3))
```

La función `elige` regresa `expt`, mientras que las llamadas recursivas cazan con `(number? sexp)` y simplemente las convierte a números:

Ejemplo

`(parse '{pow 2 3})`

`(symbol? '{pow 2 3})` NO.

`(number? '{pow 2 3})` NO.

`(list? '{pow 2 3})` Sí. Entonces, entramos al `cond` con `(car sexp) = 'pow`

Caza con `(+ * - / % max min pow)`, entonces:

```
(bin-op (elige 'pow) (parse 2) (parse 3))
```

La función `elige` regresa `expt`, mientras que las llamadas recursivas cazan con `(number? sexp)` y simplemente las convierte a números:

```
(bin-op expt (num 2) (num 3))
```

Ejercicios

Los ejercicios se encuentran en
`sites.ciencias.unam.mx/ldp171/practicas`

Enviar al correo `manu+ldp@ciencias.unam.mx` **un archivo** `equipo_sesion5.rkt` **con asunto** `[LDP-Sesión 5]` **a más tardar a las 18:59:59.**

Incluir el nombre de los integrantes en el cuerpo del correo.

Sólo pueden entregar aquellos alumnos que aparezcan en la lista de asistencia de la sesión. No es válido apuntar a miembros del equipo que no estén presentes.