Semántica de los Conceptos Fundamentales de los Lenguajes de Programación

Profesor: Juan Francisco Diaz
jdiaz@eisc.univalle.edu.co

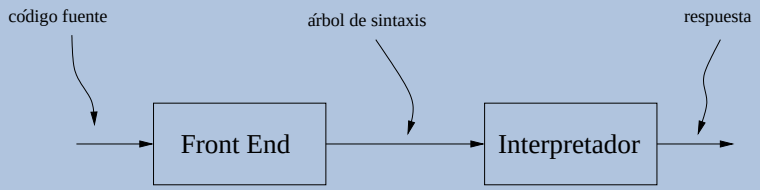
Asistente de Docencia: Gerardo M. Sarria M.
gsarria@eisc.univalle.edu.co

May 3, 2002

Interpretación y Compilación

Un interpretador es un programa que toma un árbol de sintaxis abstracta y lo convierte (posiblemente usando entradas externas) en una respuesta. El árbol es construído a partir de un programa en código fuente y pasado por un *front end*.

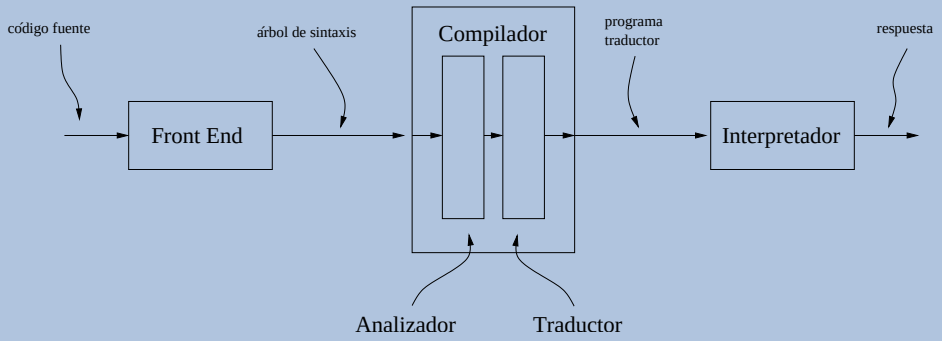
La siguiente figura representa la configuración usada para un interpretador.



Interpretación y Compilación (cont.)

Un compilador es un programa que traduce el árbol de sintaxis abstracta en otro lenguaje (el lenguaje destino), el cual es ejecutado por un interpretador.
La mayoría de veces, este otro lenguaje es un lenguaje de máquina, el cual es interpretado por un hardware; sin embargo, otras implementaciones de lenguaje usan un lenguaje destino de propósito especial que es más simple que el original y para el cual es relativamente fácil escribir un interpretador (código intermedio u objeto). Ésto último permite que el programa pueda ser compilado una vez y ejecutado en diferentes plataformas.

Un compilador está dividido en dos partes: un analizador que intenta deducir información útil del programa, y un traductor que usa la información del analizador.
La siguiente figura muestra la configuración de un compilador.



Un Interpretador Simple

Una parte importante de la especificación de cualquier lenguaje de programación es el conjunto de valores que éste manipula.
Cada lenguaje tiene como mínimo dos conjuntos:

- Los *valores expresados*: posibles valores de expresiones, y
- Los *valores denotados*: valores limitados a las variables.

Para el primer lenguaje que se creará, se tiene que

$$\begin{array}{lcl} \text{Valor Expresado} & = & \text{Número} \\ \text{Valor Denotado} & = & \text{Número} \end{array}$$

Un Interpretador Simple (cont.)

Se necesita también distinguir dos lenguajes:

- El *lenguaje definido* (o *lenguaje fuente*): es el lenguaje que se va a especificar, y
- El *lenguaje de definición* (o *lenguaje huésped*): es el lenguaje en el cual se escribirá el interpretador.

Para este caso, el lenguaje de definición es Scheme con `define-datatype` y `cases`.

Se empezará con la siguiente sintaxis:

⟨programa⟩ ::= ⟨expresión⟩
`a-program (exp)`

⟨expresión⟩ ::= ⟨número⟩
`lit-exp (datum)`

::= ⟨identificador⟩
`var-exp (id)`

::= ⟨primitiva⟩ ({⟨expresión⟩}$^{*(,)}$)
`primapp-exp (prim rands)`

⟨primitiva⟩ ::= `+` | `-` | `*` | `add1` | `sub1`

Un Interpretador Simple (cont.)

La sintaxis abstracta está construida según la gramática definida anteriormente.

```
(define-datatype program program?
  (a-program
    (exp expression?)))

(define-datatype expression expression?
  (lit-exp
    (datum number?))
  (var-exp
    (id symbol?))
  (primapp-exp
    (prim primitive?)
    (rands (list-of expression?))))

(define-datatype primitive primitive?
  (add-prim)
  (substract-prim)
  (mult-prim)
  (incr-prim)
  (decr-prim))
```

Un Interpretador Simple (cont.)

El interpretador simple consta entonces de tres procedimientos correspondientes a los tres símbolos no terminales de la gramática.

El procedimiento principal `eval-program`, toma un árbol de sintaxis abstracta y retorna un valor.
Como un programa consiste de una expresión, se usa `cases` para extraer la expresión del árbol y pasársela a `eval-expression`, junto con el ambiente inicial en el cual encontrar los valores de los identificadores de la expresión (aquí se usa el procedimiento auxiliar `init-env`)

```
(define eval-program
  (lambda (pgm)
    (cases program pgm
      (a-program (body)
        (eval-expression body (init-env))))))

(define init-env
  (lambda ()
    (extend-env
      '(i v x)
      '(1 5 10)
      (empty-env))))
```

Un Interpretador Simple (cont.)

El procedimiento `eval-expression` toma una expresión `exp` y un ambiente `env`, y retorna el valor de la expresión usando dicho ambiente para encontrar los valores de las variables.
Los casos en este procedimiento son los siguientes:

- Si `exp` es un literal, el dato es retornado.
- Si `exp` es un nodo que representa una variable, se busca el identificador en el ambiente para retornar su valor.
- Si `exp` es un nodo que representa una aplicación de una operación primitiva a algunos operandos, primero se evalúan los operandos (usando el procedimiento auxiliar `eval-rands`) y luego se pasan al procedimiento `apply-primitive` para determinar el valor.

```
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      (lit-exp (datum) datum)
      (var-exp (id) (apply-env env id))
      (primapp-exp (prim rands)
        (let ((args (eval-rands rands env)))
          (apply-primitive prim args))))))
```

Un Interpretador Simple (cont.)

El procedimiento auxiliar `eval-rands` toma una lista de operandos y un ambiente y evalúa cada operando usando `eval-rand` (el cual llama a `eval-expression` con el ambiente actual para determinar los valores de las variables).

```
(define eval-rands
  (lambda (rands env)
    (map (lambda (x) (eval-rand x env)) rands)))

(define eval-rand
  (lambda (rand env)
    (eval-expression rand env)))
```

Un Interpretador Simple (cont.)

El procedimiento `apply-primitive` toma una operación primitiva y una lista de valores, y produce el valor que se debe obtener al aplicar la operación primitiva a la lista de valores.

```
(define apply-primitive
  (lambda (prim args)
    (cases primitive prim
      (add-prim () (+ (car args) (cadr args)))
      (substract-prim () (- (car args) (cadr args)))
      (mult-prim () (* (car args) (cadr args)))
      (incr-prim () (+ (car args) 1))
      (decr-prim () (- (car args) 1)))))
```

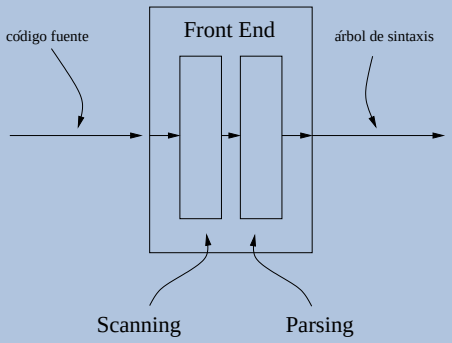
Es de notar que no se necesita pasar el ambiente como argumento a `apply-primitive` ya que éste solo trabaja con valores y no con expresiones que puedan contener variables.

El Front End

Antes de poder usar el interpretador, se necesita un *front end* que convierta el código fuente en árbol de sintaxis abstracta. Este proceso esta dividido en dos partes:

- *Scanning*: Proceso en el cual se divide el texto de entrada en *tokens* (unidades léxicas o lexemas del lenguaje).
- *Parsing*: Proceso en el cual se organizan los tokens en una estructura sintáctica jerárquica (como expresiones, declaraciones y bloques), dando como resultado una árbol de sintaxis abstracta.

La siguiente figura muestra la estructura del *front end*



El Front End (cont.)

Scanning

El trabajo de un *scanner* es recorrer la entrada (una *especificación léxica*[a]) y analizarla para producir tokens.

La siguiente figura muestra un segmento de código y la forma como el *scanner* lo analiza.

[a] La especificación léxica es una parte de la especificación del lenguaje que provee información como:

- Cualquier secuencia de espacios y nueva línea es equivalente a un solo espacio.
- Un comentario comienza con % y continúa hasta el final de la línea.
- Un identificador es una secuencia de letras y números, que comienza con una letra.

El Front End (cont.)

El lenguaje más común para escribir una especificación léxica es el lenguaje de *expresiones regulares*. La definición de este lenguaje es la siguiente:

$$\langle \mathrm{R} \rangle ::= \langle \text{caracter} \rangle \mid \langle \mathrm{R} \rangle \ \langle \mathrm{R} \rangle \mid \langle \mathrm{R} \rangle \cup \langle \mathrm{R} \rangle \mid \langle \mathrm{R} \rangle^* \mid \neg \langle \text{caracter} \rangle$$

La especificación del ejemplo de la figura anterior puede ser escrita usando expresiones regulares así:

$$\begin{array}{lcl} \langle \text{espacio-en-blanco} \rangle & = & (\langle \text{espacio} \rangle \cup \langle \text{nueva-línea} \rangle) \\ & & (\langle \text{espacio} \rangle \cup \langle \text{nueva-línea} \rangle)^* \\ \langle \text{comentario} \rangle & = & \texttt{\%}\ (\neg \langle \text{nueva-línea} \rangle)^* \\ \langle \text{identificador} \rangle & = & \langle \text{letra} \rangle\ (\langle \text{letra} \rangle \cup \langle \text{dígito} \rangle)^* \end{array}$$

El Front End (cont.)

Cuando el *scanner* encuentra un token, retorna una estructura de datos que consiste de al menos los siguientes datos:

- Una *clase*, la cual describe qué clase de token se encontró.
- Un dato que describe el token particular. Por ejemplo, para identificadores, el dato es un símbolo de Scheme contruído de la cadena en el token; para números, el dato es el número descrito por el literal; y para cadenas, el dato es la cadena.
- Un dato que describe la ubicación del token en la entrada (que sirve para ayudar al programador a identificar dónde se encuentran los errores de sintaxis).

El conjunto de clases y la descripción de los tokens hacen parte de la especificación léxica.

El Front End (cont.)

Parsing

Como se vió anteriormente, *parsing* es el proceso de organizar una secuencia de tokens en una estructura sintáctica jerárquica (definida usando una BNF y llamada *gramática libre de contexto*) como expresiones, declaraciones y bloques.

Los tipos de datos de la gramática (con los cuales se basa un *parser* para generar el árbol de sintaxis) pueden ser descritos usando `define-datatype`.
Dada una gramática, debe haber un tipo de dato para cada símbolo no terminal; y debe haber una variante por cada producción que tenga símbolor no terminales en el lado derecho. Cada variante tendrá un campo por cada símbolo no terminal, identificador, o número que aparezca en su lado derecho.

El Front End (cont.)

Ejemplo 1:
Dada la gramática

⟨declaración⟩	::=	{⟨declaración⟩ `;` ⟨declaración⟩}
	::=	`while` ⟨expresión⟩ `do` ⟨declaración⟩
	::=	⟨identificador⟩ `:=` ⟨expresión⟩
⟨expresión⟩	::=	⟨identificador⟩
	::=	(⟨expresión⟩ `+` ⟨expresión⟩)

El tipo de dato para esta gramática puede ser descrito de la siguiente manera:

```
(define-datatype statement statement?
  (compound-statement
    (stmt1 statement?)
    (stmt2 statement?))
  (while-statement
    (test expression?)
    (body statement?))
  (assign-statement
    (lhs symbol?)
    (rhs expression?)))
```

El Front End (cont.)

```
(define-datatype expression expression?
  (var-exp
    (is symbol?))
  (sum-exp
    (exp1 expression?)
    (exp2 expression?)))
```

Entonces, para la entrada

```
x := foo; while x do x := (x + bar)
```

se producirá la salida

```
(compound-statement
  (assign-statement x (var-exp foo))
  (while-statement (var-exp x)
    (assign-statement x
      (sum-expression (var-exp x) (var-exp bar)))))
```

SLLGEN

SLLGEN es un generador de *parsers* que toma como entrada una especificación léxica y una gramática, y produce como salida, un *scanner* y un *parser* en Scheme.

La especificación léxica en SLLGEN es una lista que satisface la siguiente gramática:

⟨scanner-spec⟩	::=	({⟨exp-reg-y-acción⟩}*)
⟨exp-reg-y-acción⟩	::=	(⟨nombre⟩ ({⟨exp-reg⟩}*) ⟨salida⟩
⟨nombre⟩	::=	⟨símbolo⟩
⟨exp-reg⟩	::=	⟨cadena⟩ | `letter` | `digit` | `whitespace` | `any`
	::=	(`not` ⟨caracter⟩) | (`or` {⟨exp-reg⟩}*)
	::=	(`arbno` ⟨exp-reg⟩) | (`concat` {⟨exp-reg⟩}*)
⟨salida⟩	::=	`skip` | `symbol` | `number` | `string`

SLLGEN (cont.)

A medida que el *scanner* trabaja, va recolectando caracteres en un búfer. Cuando el *scanner* determina que ha encontrado la cadena más larga posible de todas las expresiones regulares en la especificación, ejecuta la *salida* de la expresión regular correspondiente.
Dicha salida puede ser:

- `skip`: Significa que es el final de un token, pero ningún token es emitido. El *scanner* continúa trabajando en la cadena para encontrar el siguiente token. Esta acción es usada en espacios en blanco y comentarios.
- `symbol`: Los caracteres en el búfer son convertidos en un símbolo de Scheme y un token es emitido, con el nombre la clase como su clase y con el símbolo como dato.
- `number`: Los caracteres en el búfer son convertidos en un número de Scheme y un token es emitido, con el nombre la clase como su clase y con el número como dato.
- `string`: Los caracteres en el búfer son convertidos en una cadena de Scheme y un token es emitido, con el nombre la clase como su clase y con la cadena como dato.

SLLGEN (cont.)

La especificación léxica del interpretador que se está realizando será entonces:

```
(define scanner-spec-3-1
  '((white-sp
     (whitespace)                              skip)
    (comment
     ("%" (arbno (not #\newline)))             skip)
    (identifier
     (letter (arbno (or letter digit "?"))) symbol)
    (number
     (digit (arbno digit))                     number)))
```

SLLGEN (cont.)

SLLGEN incluye también una lenguaje para especificar gramáticas.
Una gramática en SLLGEN es una lista descrita por la siguiente gramática:

⟨gramática⟩	::=	({⟨producción⟩}*)
⟨producción⟩	::=	(⟨lhs⟩ ({⟨rhs-item⟩}*) ⟨nombre-prod⟩)
⟨lhs⟩	::=	⟨símbolo⟩
⟨rhs-item⟩	::=	⟨símbolo⟩ | ⟨cadena⟩
	::=	(`arbno` {⟨rhs-item⟩}*)
	::=	(`separated-list` {⟨rhs-item⟩}* ⟨cadena⟩)
⟨nombre-prod⟩	::=	⟨símbolo⟩

En SLLGEN, la gramática debe permitir al *parser* determinar cuál producción usar conociendo solo

1. qué símbolo no terminal se está buscando, y
2. el primer símbolo (token) de la cadena a ser analizada.

Las gramáticas en esta forma son denominadas *gramáticas LL* (de allí el nombre SLLGEN - Scheme LL GENerator).

SLLGEN (cont.)

La gramática del ejemplo 1 puede ser escrito en SLLGEN así:

```
(define grammar-a1
  '((statement
      ("" statement ";" statement "")
      compound-statement)
    (statement
      ("while" expression "do" statement)
      while-statement)
    (statement
      (identifier ":=" expression)
      assign-statement)
    (expression
      (identifier)
      var-exp)
    (expression
      ("(" expression "+" expression ")")
      sum-exp)))
```

SLLGEN (cont.)

La gramática del interpretador que se está realizando será entonces:

```
(define grammar-3-1
  '((program (expression)
      a-program)
    (expression (number)
      lit-exp)
    (expression (id)
      var-exp)
    (expression
      (primitive "(" (separated-list expression ",") ")" )
      primapp-exp)
    (primitive ("+")
      add-prim)
    (primitive ("-")
      substract-prim)
    (primitive ("*")
      mult-prim)
    (primitive ("add1")
      incr-prim)
    (primitive ("sub1")
      decr-prim)))
```

SLLGEN (cont.)

SLLGEN incluye muchos procedimientos para incorporar los *scanners* y gramáticas en un *parser* ejecutable.

El procedimiento `sllgen:make-define-datatypes` genera cada una de las expresiones `define-datatype` de la gramática para ser usada por `cases`.
Para el caso del interpretador, simplemente se hace un llamado a éste procedimiento:

```
(sllgen:make-define-datatypes scanner-spec-3-1 grammar-3-1)
```

SLLGEN (cont.)

El procedimiento de SLLGEN `sllgen:make-string-parser` es usado para construir un *scanner* y un *parser* basados en las especificaciones léxicas y gramaticales.
Este procedimiento retorna un procedimiento que toma una cadena y produce un árbol de sintaxis abstracta.

Se usará el procedimiento `scan&parse` para hacer el llamado a `sllgen:make-string-parser`:

```
(define scan&parse
  (sllgen:make-string-parser
    scanner-spec-3-1
    grammar-3-1))
```

Por ejemplo, si se llama a `scan&parse` con `add1(2)` como argumento, éste retorna el árbol de sintaxis abstracta correspondiente:

```
> (scan&parse "add1(2)")
(a-program (primapp-exp (incr-prim) ((lit-exp 2))))
```

SLLGEN (cont.)

Se definirá un procedimiento `run` que toma como argumento una cadena y retorna el resultado de evaluar dicha cadena usando su árbol de sintaxis.

```
(define run
  (lamba (string)
    (eval-program
      (scan&parse string))))
```

Entonces, si se llama a `run` con el mismo argumento que el ejemplo anterior (`add1(2)`), ahora devolverá el resultado y no el árbol:

```
> (run "add1(2)")
3
```

SLLGEN (cont.)

La interfaz interactiva del usuario provista por la mayoría de implementaciones de Scheme es un *read-eval-print-loop*, es decir, un ciclo que repite la acción de leer una expresión o definición, evaluarla e imprimir el resultado.

SLLGEN puede ser usado para construir un *read-eval-print-loop*, usando los siguientes procedimientos:

- `sllgen:make-stream-parser`: Toma un flujo de caracteres y genera un flujo de tokens, y
- `sllgen:make-rep-loop`: Toma una cadena `str`, un procedimiento de un solo argumento `pro` y un flujo de tokens, y produce un *read-eval-print-loop* que crea a `str` como indicador en la salida estándar, lee el flujo de tokens, los analiza, imprime el resultado de aplicar el procedimiento `pro` al árbol de sintaxis abstracta resultante, y se llama recursivamente.

SLLGEN (cont.)

Para el interpretador el procedimiento `read-eval-print` será:

```
(define read-eval-print
  (sllgen:make-rep-loop "-->" eval-program
    (sllgen:make-stream-parser
      scanner-spec-3-1
      grammar-3-1)))
```

Si se llama el procedimiento `read-eval-print`, un indicador (*listener* o *prompt*) del estilo "`-->`" quedará escuchando cualquier entrada para ser interpretada.
Por ejemplo,

```
> (read-eval-print)
--> 5
5
--> add1(2)
3
--> +(add1(2), -(6,4))
5
```

Evaluación de Condicionales

Para estudiar la semántica e implementaciones de una gran cantidad de características de un lenguaje de programación, se debe empezar por agregar dichas características al lenguaje que se está definiendo.

Para cada característica, se agregará una producción a la gramática de ⟨expresión⟩, especificando una sintaxis abstracta para cada producción, y agregando los `cases` apropiados a `eval-expression`.

Se agregará primero una expresión condicional

⟨expresión⟩ ::= `if` ⟨expresión⟩ `then` ⟨expresión⟩ `else` ⟨expresión⟩
`if-exp (test-exp true-exp false-exp)`

Evaluación de Condicionales (cont.)

Para no tener que definir un nuevo tipo de dato que maneje booleanos, falso se represtará con cero y cualquier otro valor representará verdadero (como en el lenguaje de programación C).
El procedimiento `true-value?` abstrae esta decisión:

```
(define true-value?
  (lambda (x)
    (not (zero? x))))
```

El comportamiento de los condicionales en el interpretador, se obtiene agregando la siguiente cláusula en `eval-expression`:

```
(if-exp (test-exp true-exp false-exp)
  (if (true-value? (eval-expression test-exp env))
    (eval-expression true-exp env)
    (eval-expression false-exp env)))
```

Un ejemplo del interpretador corriendo con condicionales es:

```
--> if 1 then 2 else 3
2
--> if -(3,+(1,2)) then 2 else 3
3
```