Fundamentos de lenguajes de programación

Semántica de los Conceptos Fundamentales de Lenguajes de Programación

Facultad de Ingeniería. Universidad del Valle

Septiembre de 2022

Contenido

Contenido

Contenido

Contenido

Contenido

Contenido

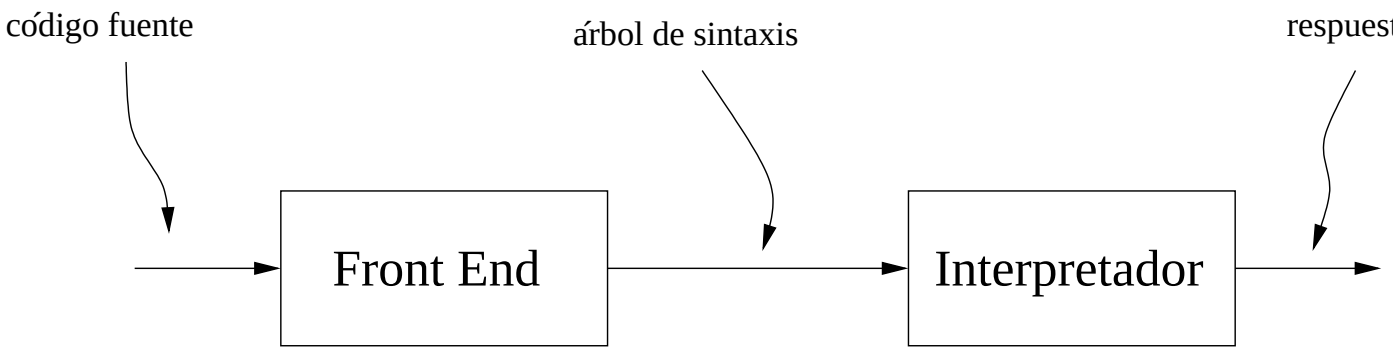
Contenido

Interpretación

- El texto de un programa es escrito en un lenguaje llamado el *lenguaje fuente* o el *lenguaje definido*.
- Los programas son pasados a través de un *front end* que los analiza y construye el árbol de sintaxis abstracta.
- Luego, el árbol de sintaxis abstracta es pasado a un interpretador, que examina su estructura y desarrolla algunas acciones que dependen de esa estructura.

Interpretación

La siguiente figura ilustra el proceso llevado a cabo cuando se utiliza un interpretador:



Interpretación

- Un interpretador es un programa que toma un árbol de sintaxis abstracta y lo convierte en una respuesta.
- Un interpretador está escrito en algún lenguaje. Este lenguaje es llamado el *lenguaje de implementación* o el *lenguaje de definición*.

Interpretación

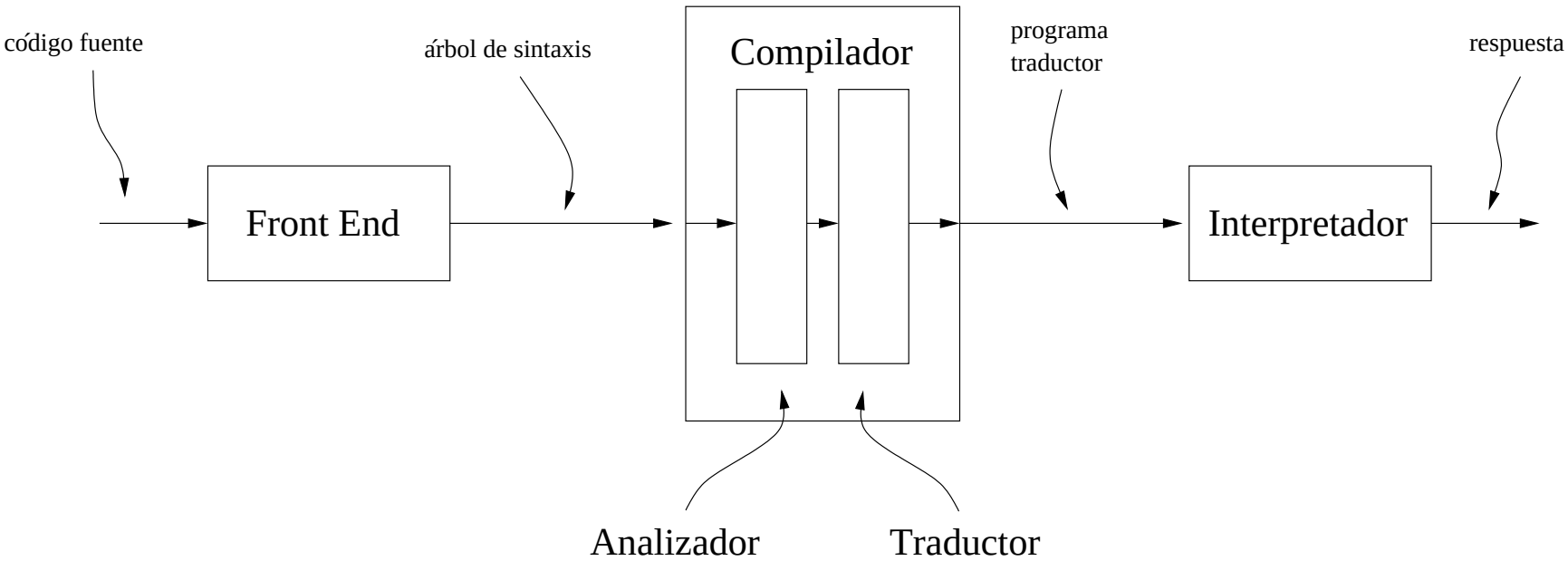
- Los programas interpretados suelen ser más lentos que los programas compilados debido a que es necesario traducir el programa mientras se ejecuta.
- Los lenguajes de programación interpretados son más flexibles y favorecen la modularidad. Por esta razón se logra mayor velocidad de desarrollo con su uso.

Compilación

- Otra alternativa para el análisis y ejecución de programas es el uso de un compilador.
- Un compilador traduce el árbol de sintaxis abstracta en un programa en otro lenguaje para ser ejecutado. Este lenguaje es llamado el *lenguaje destino*.
- El lenguaje generado puede ser ejecutado por un interpretador o puede ser traducido en un lenguaje de bajo nivel para su ejecución.

Compilación

La siguiente figura ilustra el proceso llevado a cabo cuando se utiliza un compilador:



Compilación

- Por lo general, el lenguaje destino es un lenguaje máquina interpretado por un hardware.
- 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).
- Esto permite que el programa pueda ser compilado una vez y ejecutado en diferentes plataformas.
- Este tipo de lenguajes destino son llamados *lenguajes a bytecode* y sus interpretadores *máquinas virtuales*.

Compilación

- Un compilador está dividido en dos partes: un *analizador* y un *traductor*.
- El analizador tiene como finalidad deducir información útil sobre el programa.
- El traductor lleva a cabo la traducción del programa a partir de la información del analizador.

Interpretación y Compilación

- Sin importar, la estrategia que se utilice, es neceario definir un *front end* que convierta programas en árboles de sintaxis abstracta.
- Dado que los programas son solo cadenas de caracteres, el front end debe agrupar estos caracteres en unidades significativas.
- La agrupación de estas unidades es llevada a cabo en dos etapas: *scanning* y *parsing*.

Interpretación y Compilación

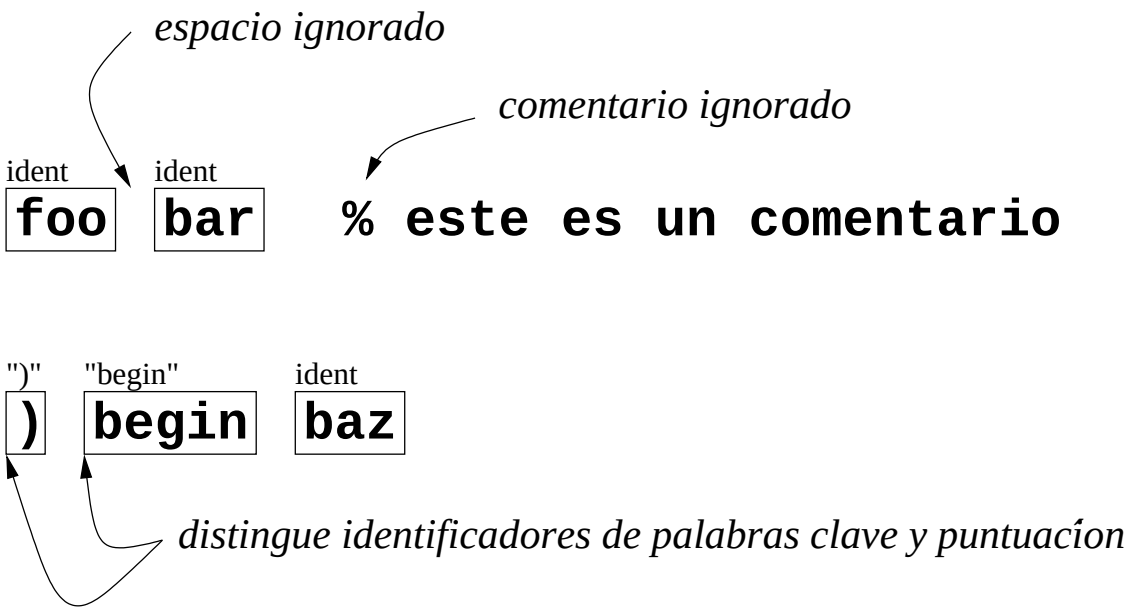
Scanning

- *Scanning* es el nombre que se le da al proceso de dividir la secuencia de caracteres en palabras, números, puntuación, comentarios, etc.
- Estas unidades son conocidas como *unidades léxicas*, *lexemas* o *tokens*.
- La *especificación léxica* de un lenguaje se refiere a la forma en la cual un programa debe ser dividido en unidades léxicas.
- El *scanner* recibe una secuencia de caracteres y produce una secuencia de unidades léxicas (tokens).

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.

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



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.

- *Parsing* es el nombre que se le da al proceso de organizar una secuencia de tokens en estructuras sintácticas jerárquicas como expresiones, estamentos y bloques.
- La estructura *sintáctica* o gramátical de un lenguaje se refiere a la forma en la cual se deben organizar las unidades léxicas.
- El *parser* recibe una secuencia de tokens del *scanner* y produce un árbol de sintaxis abstracta.

- Un generador de *parser* es un programa que toma como entrada una especificación léxica y una gramática y produce como salida un *scanner* y un *parser* para ellos.
- 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 símbolos 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.

Contenido

SLLGEN

Especificación Léxica

- 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`

- 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 *symbol* como su clase y con el símbolo como dato.

Dicha salida puede ser:

- `number`: Los caracteres en el búfer son convertidos en un número de Scheme y un token es emitido, con el nombre *number* 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 *string* como su clase y con la cadena como dato.

SLLGEN

Gramáticas

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

$$
\begin{array}{lcl}
\langle \text{gramática} \rangle & ::= & (\{\langle \text{producción} \rangle\}^*) \\
\langle \text{producción} \rangle & ::= & (\langle \text{lhs} \rangle\ (\{\langle \text{rhs-item} \rangle\}^*)\ \langle \text{nombre-prod} \rangle) \\
\langle \text{lhs} \rangle & ::= & \langle \text{símbolo} \rangle \\
\langle \text{rhs-item} \rangle & ::= & \langle \text{símbolo} \rangle \mid \langle \text{cadena} \rangle \\
 & ::= & (\texttt{arbno}\ \{\langle \text{rhs-item} \rangle\}^*) \\
 & ::= & (\texttt{separated-list}\ \{\langle \text{rhs-item} \rangle\}^*\ \langle \text{cadena} \rangle) \\
\langle \text{nombre-prod} \rangle & ::= & \langle \text{símbolo} \rangle
\end{array}
$$

SLLGEN

Gramáticas

- 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

Expresiones `define-datatype`

- SLLGEN incluye 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`.
- El procedimiento `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.

SLLGEN

Interfaz Interactiva

- 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.

Ejemplo:

Dada la gramática:

$$
\begin{array}{lcl}
\langle \text{declaración} \rangle & ::= & \{\langle \text{declaración} \rangle \texttt{ ; } \langle \text{declaración} \rangle\} \\
 & ::= & \texttt{while } \langle \text{expresión} \rangle \texttt{ do } \langle \text{declaración} \rangle \\
 & ::= & \langle \text{identificador} \rangle \texttt{ := } \langle \text{expresión} \rangle \\
\langle \text{expresión} \rangle & ::= & \langle \text{identificador} \rangle \\
 & ::= & (\langle \text{expresión} \rangle \texttt{ + } \langle \text{expresión} \rangle)
\end{array}
$$

Ejemplo:

Los tipos de datos para está gramática pueden ser descritos así:

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

Ejemplo:

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

Ejemplo:

Los especificación léxica para el interpretador será:

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

Ejemplo:

La gramática puede ser escrita en SLLGEN de la siguiente manera:

```
(define grammar
  '((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)))
```

Ejemplo:

Para generar las expresiones `define-datatype` se utiliza:

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

Para crear el *scanner* y el *parser* se tiene:

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

Ejemplo:

Si se llama a `scan&parse` con `x := y` como argumento, éste retorna el árbol de sintaxis abstracta correspondiente:

```
> (scan&parse "x := y")
(assign-statement (x (var-exp (y))))
```

Ejemplo:

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
      grammar)))
```

Contenido

Un Interpretador Simple

- Nuestro primer lenguaje permitirá la evalución de expresiones aritméticas.
- El lenguaje consistirá de expresiones para variables, números y aplicación de los operadores $+$, *, -. `add1` y `sub1`.

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

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

Un Interpretador Simple

Gramática

La gramática para el lenguaje será la siguiente:

⟨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

Especificación Léxica

La especificación léxica para el lenguaje será la siguiente:

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

Un Interpretador Simple

Especificación de la Gramática

La especificación de la gramática es la siguiente:

```
(define grammar-simple-interpreter
  '((program (expression) a-program)
    (expression (number) lit-exp)
    (expression (identifier) var-exp)
    (expression
       (primitive "(" (separated-list expression ",")")")
        primapp-exp)
    (primitive ("+") add-prim)
    (primitive ("-") subtract-prim)
    (primitive ("*") mult-prim)
    (primitive ("add1") incr-prim)
    (primitive ("sub1") decr-prim)
))
```

Un Interpretador Simple

Sintaxis Abstracta

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?))))
```

Un Interpretador Simple

Sintaxis Abstracta

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

Estas definiciones pueden ser generadas automáticamente mediante SLLGEN con el procedimiento `sllgen:make-define-datatypes`

Un Interpretador Simple

- El interpretador simple constará 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.

Un Interpretador Simple

```
(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

- 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.

Un Interpretador Simple

```
(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

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

- 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.

Contenido

Un interpretador más complejo

- Nuestro lenguaje será extendido para incorporar condicionales y ligadura local (asignación).
- El lenguaje consistirá de las expresiones especificadas anteriormente y de expresiones para condicionales `if ... then ... else` y para el operador de ligadura local `let`.
- Para este lenguaje se extiende el conjunto de valores expresados y denotados de la siguiente manera:

$$\begin{aligned} \text{Valor Expresado} &= \text{Número} + \text{Booleano} \\ \text{Valor Denotado} &= \text{Número} + \text{Booleano} \end{aligned}$$

Un interpretador más complejo

Gramática

La gramática para el lenguaje será la siguiente:

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

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

::= "true"
`true-exp`

::= "false"
`false-exp`

::= ⟨identificador⟩
`var-exp (id)`

Un interpretador más complejo

Gramática

⟨primitiva⟩ (⟨expresión⟩)*(,)

```
primapp-exp (prim rands)
```

if ⟨expresión⟩ then ⟨expresión⟩ else ⟨expresión⟩

```
if-exp (test-exp true-exp false-exp)
```

let ⟨identificador⟩ = ⟨expresión⟩* in ⟨expresión⟩

```
let-exp (ids rands body)
```

⟨primitiva⟩ ::= + | - | * | add1 | sub1 | == | ¿= | ¡= | ¿ | ¡

Un interpretador más complejo

Especificación Léxica

La especificación léxica será la misma del lenguaje anterior:

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

Un interpretador más complejo

Especificación de la Gramática

La especificación de la gramática es la siguiente:

```
(define grammar-simple-interpreter
  '((program (expression) a-program)
    (expression (number) lit-exp)
    (expression (identifier) var-exp)
    (expression ("true") true-exp)
    (expression ("false") false-exp)
    (expression (primitive "(" (separated-list expression ",
        ")")")")
        primapp-exp)
    (expression ("if" expression "then" expression "else"
        expression)
        if-exp)
    (expression ("let" (arbno identifier "=" expression) "in
        " expression)
        let-exp)
    (primitive ("+") add-prim)
    (primitive ("-") subtract-prim)
    (primitive ("*") mult-prim)
    (primitive ("add1") incr-prim)
    (primitive ("sub1") decr-prim)
))
```

Un interpretador más complejo

Sintaxis Abstracta

La sintaxis abstracta está construida de la siguiente manera.

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

Un interpretador más complejo

Sintaxis Abstracta

```
(define-datatype expression expression?
  (lit-exp
    (datum number?))
  (var-exp
    (id symbol?))
  (primapp-exp
    (prim primitive?)
    (rands (list-of expression?)))
  (if-exp
    (test-exp expression?)
    (true-exp expression?)
    (false-exp expression?))
  (let-exp
    (ids (list-of symbol?))
    (rans (list-of expression?))
    (body expression?)))
```

Un interpretador más complejo

Sintaxis Abstracta

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

Contenido

Semántica de los condicionales

- Para determinar el valor de una expresión condicional (`if-exp` exp_1 exp_2 exp_3) es necesario determinar el valor de la subexpresión exp_1.
- Si este valor corresponde al valor booleano *true*, el valor de toda la expresión `if-exp` debe ser el valor de la subexpresión exp_2.
- En caso contrario, el valor de la expresión `if-exp` debe ser el valor de la subexpresión exp_3.

Semántica de los condicionales

- Para poder determinar si el valor de una expresión es un valor booleano (verdadero o falso), es necesario dar alguna representación a este tipo de dato.
- Para esto se define el tipo de dato booleano, como true y false. Además se agregan las primitivas booleanas de comparación.

Semántica de los condicionales

De esta manera, el comportamiento de los condicionales en el interpretador, se obtiene agregando la siguiente clausula en el procedimiento `eval-expression`:

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

Semántica de los condicionales

Ejemplo

- Sea el ambiente env_0 con símbolos (x y z) y valores (4 2 5) el ambiente inicial de computación.
- Se quiere evaluar la expresión:

```
if >(x,4) then +(y,11) else *(y,10)
```

- Primero se evalua la subexpresión `>(x,4)`. Dado que x vale 4, este da falso.
- Como el valor de la subexpresión`>(x,4)` es falso, se evalúa la subexpresión `*(y,10)`.
- Finalmente, el valor de toda la expresión `if` es 20.

Semántica de los condicionales

Ejemplo

- Sea el ambiente env_0 con símbolos (x y z) y valores (4 2 5) el ambiente inicial de computación.
- Se quiere evaluar la expresión:

```
if >(x,4) then +(y,11) else *(y,10)
```

- Primero se evalua la subexpresión `>(x,4)`. Dado que x vale 4, este da falso.
- Como el valor de la subexpresión`>(x,4)` es falso, se evalúa la subexpresión `*(y,10)`.
- Finalmente, el valor de toda la expresión `if` es 20.

Contenido

Semántica de la ligadura local

- Hasta el momento, todas las expresiones del lenguaje se evalúan en el mismo ambiente (el ambiente inicial).
- La expresión `let` permite la creación de ligaduras locales a variables nuevas.
- Típicamente, la expresión `let` crea un nuevo ambiente que extiende el ambiente principal (sobre el que se evalúa el `let`) con las variables y valores especificados en el contenido de la expresión.

Semántica de la ligadura local

- Para determinar el valor de una expresión (`let-exp` *ids exps body*) es necesario evaluar las partes derechas de las declaraciones (correspondientes a las expresiones `exps`) en el ambiente anterior.
- Posteriormente, debe crearse un nuevo ambiente extendiendo el ambiente anterior con las variables de la declaración y sus valores (obtenidos al evaluar las expresiones `exps`).
- Finalmente, se evalúa la expresión `body` en el nuevo ambiente extendido.

Semántica de la ligadura local

De esta manera, el comportamiento del operador de ligadura local, se obtiene agregando la siguiente clausula en el procedimiento `eval-expression`:

```
(let-exp (ids rands body)
   (let ((args (eval-rands rands env)))
      (eval-expression body
                       (extend-env ids args env))))
```

Semántica de la ligadura local

Ejemplo

- Sea el ambiente env_0 con símbolos (x y z) y valores (4 2 5) el ambiente inicial de computación.
- Se quiere evaluar la expresión:

```
let
  x = -(y,1)
in
  let
    x = +(x,2)
  in
    add1(x)
```

Semántica de la ligadura local

Ejemplo

- Primero se evalua la subexpresión `-(y,1)` correspondiente a la parte derecha de la única declaración en el `let` exterior.
- El valor de la subexpresión `-(y,1)` es 1 por lo que se crea un ambiente env_1 que extiende el ambiente anterior env_0 con la variable x y el valor 1.
- Posteriomente se evalúa la expresión:

```
let
   x = +(x,2)
in
   add1(x)
```

en el ambiente env_1.

Semántica de la ligadura local

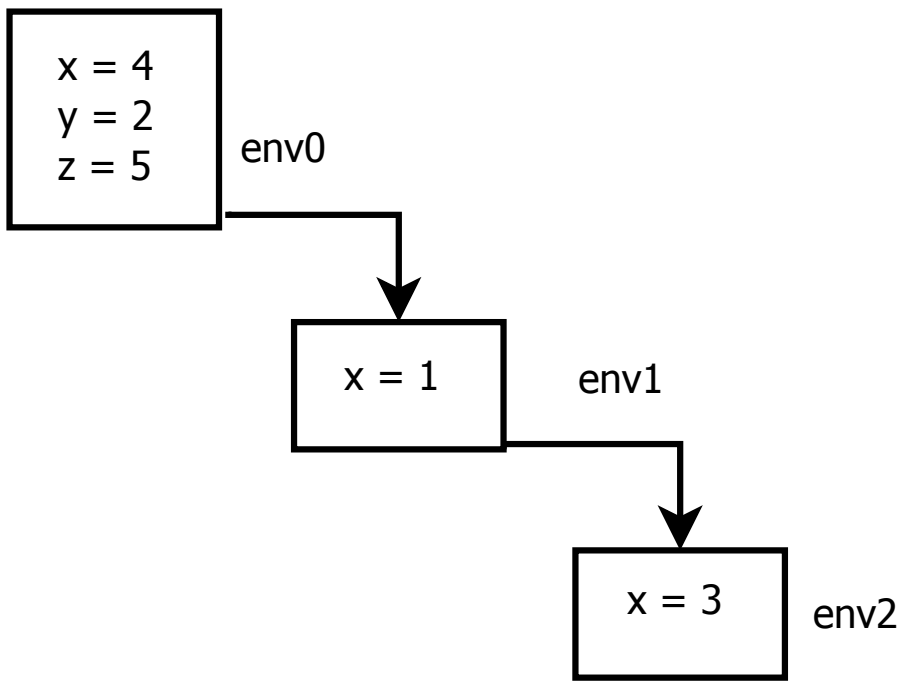
Ejemplo

- Se evalua la subexpresión `+(x,2)` correspondiente a la parte derecha de la única declaración en el `let` interior.
- El valor de la subexpresión `+(x,2)` es 3 por lo que se crea un ambiente env_2 que extiende el ambiente env_1 con la variable x y el valor 3.
- Posteriomente se evalúa la expresión `add1(x)` en el ambiente env_2.
- Finalmente, el valor de la expresión original es 4.

Semántica de la ligadura local

Ejemplo

Los ambientes creados en la evaluación de la expresión anterior se pueden visualizar así:



Interpretador

Finalmente, el procedimiento `eval-expression` se define así:

```
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      ...
      (let-exp (ids rands body)
               (let ((args (eval-rands rands env)))
                 (eval-expression body
                                  (extend-env ids args env))
                     )))))
```

Semántica de la ligadura local

Ejemplo

Sea el ambiente env_0 con símbolos (x y z) y valores (4 2 5) el ambiente inicial de computación. Evaluar:

```
let
    z=5
    t=sub1(x)
in
  let
    x= -( t , 1)
  in
    let
      y= 4
    in
      *(t , -(z , -(x,y)))
```

Semántica de la ligadura local

Ejemplo

Sea el ambiente env_0 con símbolos (x y z) y valores (4 2 5) el ambiente inicial de computación. Evaluar:

```
let
    z=5
    t= let
          p = 5
          q = 2
       in
          +(p,q)
in
  let
    x= add1(z)
  in
    *(t, *(y,x))
```

Semántica de la ligadura local

Ejercicio para entregar

Sea el ambiente env_0 con símbolos (x y z) y valores (3 7 1) el ambiente inicial de computación. Evaluar:

```
let
    y=5
    m= let
          t = sub1(y)
       in
          *(t,x)
in
  let
    y= if >(y,3) then +(add1(y), m) else sub1(+(y,m))
  in
    let
      t = -(y,m)
    in
      +(t, +(y, -(z, 3)))
```

Dibuje los ambientes creados en la evaluación de la expresión.

Preguntas

?

Próxima sesión

- Semántica de la creación y aplicación de procedimientos.