Lenguaje LET

Este es un lenguaje simple (pero no tan simple como el aritmético) que permite la definición y referencia de variables locales.

Sintaxis

La sintaxis concreta del lenguaje se especifica con una gramática independiente del contexto, las categorías sintácticas o no-terminales del lenguaje se enfatizan en itálicas, mientras que los terminales se escriben sin énfasis, o bien, en negritas para estilizar palabras reservadas. Una producción de la forma $X ::= \alpha$ establece que se puede derivar la cadena de terminales y no-terminales α a partir de la categoría sintáctica X.

La sintaxis abstracta del lenguaje se especifica a partir de la estructura de las posibles expresiones, cada una de estas se denota como una lista cuyo primer elemento es el tipo de expresión, seguido de los nombres asociados a las subestructuras inmediatas, estos nombres suelen indicar el tipo particular de la subestructura. Una estructura de la forma $(x\ y_1\ y_2\ \ldots\ y_n)$ representa la existencia de: (a) un constructor x que toma n argumentos y produce un árbol de sintaxis de este tipo; (b) un predicado $x?$ que toma un argumento y determina si es de tipo x; (c) n selectores x-y_i que toman un argumento de tipo x y regresan su subestructura correspondiente a y_i.

Se presupone la existencia de un parser que reconoce programas escritos de acuerdo a la sintaxis concreta del lenguaje que produce estructuras de acuerdo a la sintaxis abstracta del lenguaje.

Sintaxis concreta		**Sintaxis abstracta**
Expression ::=	*Number*	(const-exp *num*)
Expression ::=	**-**(*Expression* , *Expression*)	(diff-exp *exp1 exp2*)
Expression ::=	**zero?**(*Expression*)	(zero?-exp *exp1*)
Expression ::=	**if** *Expression* **then** *Expression* **else** *Expression*	(if-exp *exp1 exp2 exp3*)
Expression ::=	*Identifier*	(var-exp *var*)
Expression ::=	**let** *Identifier* = *Expression* **in** *Expression*	(let-exp *var exp1 body*)

Semántica

La interpretación de expresiones del lenguaje de acuerdo a su sintaxis abstracta se especifica utilizando semántica operacional de pasos grandes (también llamada semántica natural) que consiste de afirmaciones de la forma (value-of $e\ \rho) = v$ donde e es una expresión, ρ un entorno y v un valor expresado.

Existe un entorno vacío ρ_0, un mecanismo para extender un entorno ρ con la vinculación de una variable x a un valor v denotado como $[x = v]\rho$ y un mecanismo para aplicar entornos a variables $\rho(x)$. En particular, los entornos se modelan como funciones de variables a valores, con ρ_0 indefinida para cualquier variable.

En este lenguaje los valores expresados $ExpVal$ (valores de expresiones) y los valores denotados $DenVal$ (valores asociados en entornos) son los mismos y corresponden a $Int + Bool$. Los procedimientos expval→num y expval→bool toman valores expresados y regresan los valores codificados en el lenguaje de implementación.

Interpretación de expresiones

(value-of (const-exp n) ρ) = (num-val n)

(value-of (var-exp var) ρ) = $\rho(var)$

(value-of (diff-exp *exp1 exp2*) ρ)
  = (num-val (- (expval→num (value-of *exp1* ρ))
                (expval→num (value-of *exp2* ρ))))

(value-of (zero?-exp *exp1*) ρ)
  = (**let** ([*val1* (value-of *exp1* ρ)])
      (bool-val (= 0 (expval→num *val1*))))

(value-of (if-exp *exp1 exp2 exp3*) ρ)
  = (**if** (expval→bool (value-of *exp1* ρ))
      (value-of *exp2* ρ)
      (value-of *exp3* ρ))

(value-of (let-exp *var exp1 body*) ρ)
  = (**let** ([*val1* (value-of *exp1* ρ)])
      (value-of *body* $[var = val1]\rho$))

Lenguaje PROC

Este lenguaje está basado en LET pero contempla la creación e invocación de procedimientos.

Sintaxis

Sintaxis concreta

Expression ::=	*Number*
Expression ::=	**-**(*Expression* , *Expression*)
Expression ::=	**zero?**(*Expression*)
Expression ::=	**if** *Expression* **then** *Expression* **else** *Expression*
Expression ::=	*Identifier*
Expression ::=	**let** *Identifier* = *Expression* **in** *Expression*
Expression ::=	**proc** (*Identifier*) *Expression*
Expression ::=	(*Expression Expression*)

Sintaxis abstracta

(const-exp *num*)
(diff-exp *exp1 exp2*)
(zero?-exp *exp1*)
(if-exp *exp1 exp2 exp3*)
(var-exp *var*)
(let-exp *var exp1 body*)
(proc-exp *var body*)
(call-exp *op-exp arg-exp*)

Semántica

Los valores expresados *ExpVal* (valores de expresiones) y los valores denotados *DenVal* (valores asociados en entornos) son los mismos y corresponden a $Int + Bool + Proc$. Los procedimientos expval→num, expval→bool y expval→proc toman valores expresados y regresan los valores codificados en el lenguaje de implementación.

Interpretación de expresiones

(value-of (const-exp n) ρ) = (num-val n)

(value-of (var-exp *var*) ρ) = ρ(*var*)

(value-of (diff-exp *exp1 exp2*) ρ)
  = (num-val (- (expval→num (value-of *exp1* ρ))
                (expval→num (value-of *exp2* ρ))))

(value-of (zero?-exp *exp1*) ρ)
  = (**let** ([*val1* (value-of *exp1* ρ)])
      (bool-val (= 0 (expval→num *val1*))))

(value-of (if-exp *exp1 exp2 exp3*) ρ)
  = (**if** (expval→bool (value-of *exp1* ρ))
      (value-of *exp2* ρ)
      (value-of *exp3* ρ))

(value-of (let-exp *var exp1 body*) ρ)
  = (**let** ([*val1* (value-of *exp1* ρ)])
      (value-of *body* [*var* = *val1*]ρ))

(value-of (proc-exp *var body*) ρ)
  = (proc-val (procedure *var body* ρ))

(value-of (call-exp *op-exp arg-exp*) ρ)
  = (**let** ([*proc* (expval→proc (value-of *op-exp* ρ))]
        [*arg* (value-of *arg-exp* ρ)])
      (apply-procedure *proc arg*))

donde:
(apply-procedure (procedure *var body* ρ) *val*)
  =(value-of *body* [*var* = *val*]ρ)

Estrategias de implementación

Representación procedural:

```
(define (procedure var body ρ)
  (lambda (val)
    (value-of body [var = val]ρ)))

(define (apply-procedure proc1 val)
  (proc1 val))
```

Representación con estructuras de datos:

```
(define-datatype proc proc?
  (procedure
    (var identifier?)
    (body expression?)
    (saved-env environment?)))

(define (apply-procedure proc1 val)
  (cases proc proc1
    (procedure (var body ρ)
      (value-of body [var = val]ρ))))
```

Lenguaje LETREC

Este lenguaje está basado en PROC pero contempla la definición e invocación de procedimientos explícitamente recursivos.

Sintaxis

Sintaxis concreta

		Sintaxis abstracta
Expression ::=	*Number*	(const-exp *num*)
Expression ::=	**-**(*Expression* , *Expression*)	(diff-exp *exp1 exp2*)
Expression ::=	**zero?**(*Expression*)	(zero?-exp *exp1*)
Expression ::=	**if** *Expression* **then** *Expression* **else** *Expression*	(if-exp *exp1 exp2 exp3*)
Expression ::=	*Identifier*	(var-exp *var*)
Expression ::=	**let** *Identifier* = *Expression* **in** *Expression*	(let-exp *var exp1 body*)
Expression ::=	**proc** (*Identifier*) *Expression*	(proc-exp *var body*)
Expression ::=	(*Expression Expression*)	(call-exp *op-exp arg-exp*)
Expression ::=	**letrec** *Identifier* (*Identifier*) = *Expression* **in** *Expression*	(letrec-exp *p-name b-var p-body letrec-body*)

Semántica

Interpretación de expresiones

(value-of (const-exp n) ρ) = (num-val n)

(value-of (var-exp *var*) ρ) = ρ(*var*)

(value-of (diff-exp *exp1 exp2*) ρ)
  = (num-val (- (expval→num (value-of *exp1* ρ))
                (expval→num (value-of *exp2* ρ))))

(value-of (zero?-exp *exp1*) ρ)
  = (**let** ([*val1* (value-of *exp1* ρ)])
      (bool-val (= 0 (expval→num *val1*))))

(value-of (if-exp *exp1 exp2 exp3*) ρ)
  = (**if** (expval→bool (value-of *exp1* ρ))
      (value-of *exp2* ρ)
      (value-of *exp3* ρ))

(value-of (let-exp *var exp1 body*) ρ)
  = (**let** ([*val1* (value-of *exp1* ρ)])
      (value-of *body* [*var* = *val1*]ρ))

(value-of (proc-exp *var body*) ρ)
  = (proc-val (procedure *var body* ρ))

(value-of (call-exp *op-exp arg-exp*) ρ)
  = (**let** ([*proc* (expval→proc (value-of *op-exp* ρ))]
        [*arg* (value-of *arg-exp* ρ)])
      (apply-procedure *proc arg*))

donde:

(apply-procedure (procedure *var body* ρ) *val*)
  =(value-of *body* [*var* = *val*]ρ)

(value-of (letrec-exp *p-name b-var p-body letrec-body*) ρ)
  = (value-of *letrec-body*
      [*p-name* = *b-var* $\mapsto$ *p-body*]ρ)

donde:

Si ρ_1 = [*p-name* = *b-var* $\mapsto$ *p-body*]ρ, entonces
  (apply-env ρ_1 *var*) =
    (proc-val (procedure *b-var p-body* ρ_1))
si *var* = *p-name*, y
  (apply-env ρ_1 *var*) = (apply-env ρ *var*)
si *var* $\neq$ *p-name*.

Estrategias de implementación

Representación con estructuras de datos:

```
(define-datatype environment environment?
  (empty-env)
  (extend-env (var identifier?) (val expval?) (env environment?))
  (extend-env-rec (p-name identifier?) (b-var identifier?) (body expression?) (env environment?)))

(define (apply-env env search-var)
  (cases environment env
    (empty-env () ...señala un error...)
    (extend-env (saved-var saved-val saved-env)
      (if (eqv? saved-var search-var) saved-val
        (apply-env saved-env search-var)))
    (extend-env-rec (p-name b-var p-body env)
      (if (eqv? search-var p-name)
        (proc-val (procedure b-var p-body env))
        (apply saved-env search-var)))))
```