Universidad Nacional Autónoma De México

Lenguajes de Programación

Proyecto 1: MINILISP

Jiménez Rivera Emiliano Kaleb
Lenin Quetzal Vazquez Merino

Facultad de Ciencias

1 Introducción

1.1 Motivación

A lo largo del curso hemos visto algunos de los conceptos básicos de un compilador. Empezando por el análisis lexico, seguido por el análisis sintactico y terminando por el análisis semántico. Con la finalidad de concretar y llevar a la práctica los conocimientos adquiridos, tenemos la tarea de realizar este proyecto para asi tener una mejor comprensión y un mejor análisis de los lenguajes de programación desde un enfoque matematico.

1.2 Objetivos general

Se pretende Formalizar un conjunto reducido de Lisp(llamado MiniLisp), para ello se hará uso de los temas y herramientas proporcionados en clases, desde La sintaxis concreta, la sintaxis abstracta y la semantica operacional. Con el fin de tener una mejor idea del MiniLisp que se quiere obtener, a continuación se presenta cada una de las operaciones y constantes que estarán disponibles, además, se incluye una breve descripción del comportamiento deseado(esto en caso de los operadores)

-

1. Definir la sintaxis léxica y libre de contexto de MiniLisp, así como su representación en sintaxis abstracta, incorporando la noción de azúcar sintáctica y su eliminación hacia el nucleo.
2. Establecer la semántica operacional estructural del lenguaje, especificando máquinas abstractas como modelos conceptuales y máquinas virtuales como implementaciones ejecutable en HASKELL.
3. Modelar formalmente ambientes de evaluación y bindings como mecanismos fundamentales para la consistencia semántica y la correcta gestión de alcances y valores.
4. Integrar el régimen de evaluación ansioso con estrategia de paso de parámetros por valor(call-by-value), analizando su impacto en la ejecución de programas y el diseño del intérprete.
5. Evidenciar, mediante la eliminación de la azúcar sintáctica, el tránsito desde la expresividad del lenguaje hacia un núcleo reducido, fortaleciendo la comprensión del puente entre la teoría y la práctica en el diseño de lenguajes.

2 Formalización

La **Sintaxis Concreta** define exactamente como se deben escribir los programas en un lenguaje de programación, es decir, especifica las reglas de escrituras dentro del lenguaje. Estas especificaciones de las reglas se clasifica en **Sintaxis Léxica** y en la **Sintaxis Libre de Contexto**, que dan paso a programas bien construidos evitando ambiguedades.

2.1 Sintaxis léxica

La sintaxis léxica hace referencia a la estructura de los *tokens*, también llamados *lexemas*, que constituyen las unidades mínimas de significado dentro de un lenguaje de programación, por ejemplo, los operadores

e identificadores. Formalmente se usan **expresiones regulares** para definir éstas unidades mínimas.

A continuación se definiran los dos componentes principales de la sintaxis para nuestro Minilisp :

1. **Alfabeto** ($\sum$): es el conjunto finito de símbolos y caracteres que están permitidos en el lenguaje.

$$\sum = \{0,1,2,3,4,5,7,8,9\} \cup \{+,-,*,/\} \cup \{a-z,\_\} \cup \{A-Z\} \cup \{[,],(,),\#,!\} \cup \{,\}$$

2. **Tokens**: son secuencias de caracteres que representan a las unidades básicas de significado dentro del lenguje. Además son representados mediante expresiones regulares.

Comenzemos por definir los tokens no especificos, es decir, aquellos que no asocian algún valor:

- Parentesis, corchetes y coma:
$$(+)+[+]+,$$
- Operadores aritmeticos:
$\{+,-,*,/,--,++,\text{sqrt},\exp\}$
$$-+++-+/+*+--+++\text{sqrt}+\exp$$
- Operadores booleanos
$\{=,<,>,>=,<=,!=\}$
$$=+<+>+!=+<=+>=$$
- palabras reservadas y más
{not ,if ,fst ,snd ,let ,letsec ,head ,tail ,lambda ,app}
$$\text{not}+\text{if}+\text{fst}+\text{snd}+\text{let}+\text{letsec}+\text{head}+\text{tail}+\text{lambda}+\text{app}$$

Ahora definamos los tokens específicos, es decir, aquellos que asocian un valor

- Booleanos
"#t" + "#f"
- Numeros
Para nuestro Minilisp genraremos únicamente números enteros.
Sea $D = \{0,1,2,3,4,5,6,7,8,9\}$ y $Z = D - \{0\}$. Por lo tanto el conjunto buscado es
$$ZD^*+-ZD^*$$
- Identificadores
Los identificadores son cadenas, así que dado los conjuntos $F = \{a-z, A-Z, \_\}$ y
$S = \{a-z, A-Z, 0-9, \_\}$. Por lo que el conjunto de lo identificadores esta dado por
$$SD^*$$

Ahora veamos los aspectos más importantes de la implementación, para esto, se decidió hacer uso de Alex(una herramienta para generar analizadores léxico en haskell).

Se hizo uso de la envoltura básica, %wrapper "basic", ya que es una buena manera de generar a la funcion **lexer ::** String $->$ Token, puesto que la envoltura nos proporciona la función alexscanTokens(se encarga de generar los tokens).

Otro aspecto importante es justo **tokens : -**, ya que acá es donde se forman los tokens. Como nota importante, hay que tomar en cuenta la secuencia en la que formamos los Tokens, puesto que pueden ocurrir incongruencias si no se maneja con cuidado. Por ejemplo, si primero defino las siguientes reglas en el orden:

$$\backslash - \qquad \{\backslash\_ \rightarrow TokenResta\}$$
$$\backslash -?\$digit+ \qquad \{\backslash s \rightarrow TokenNum(read\, s)\}$$

Entonces, si analizo la cadena "-124", lo que resulta es que el lexer comienza por analizar el -, luego se genera el [TokenMenos], pero en realidad lo que debería regresar es [TokenNum (-124)], ya que para las operaciones estamos dejando un espacio, por ejemplo "(- 1 2 4)". Luego, este se resuelve fácilmente invirtiendo el orden de las reglas.

Y finalmente, no menos importante **$white** es usado para deshacerse de los espacio en blanco, lo que se realiza carácter por caracter, jugando así un papel importante. Pues esta interrupción nos permite diferenciar las expresiones de manera concisa, y así lograr generar los tokens indicados, como por ejemplo "-23" y "(- 2 3)", generando unas buenas bases para el análisis sintactico y semántico.

Conclusión: La implementación teóricamente fue bastante sencilla, sin embargo, para implementarlo en Alex se tuvieron que tomar en consideración muchas más cosas, lo que parecería algo malo, pero terminó siendo de gran beneficio ya que eso nos permitió analizar de mejor manera cada aspecto y tener una buen comienzo, evitando errores y posibles ambiguedades.

2.2 Sintaxis libre de contexto

la sintaxis libre de contexto es la que se encarga de describir formalmente como se combinan de forma correcta los componentes lexicos, para ello se hace uso la **Gramática libre de contexto**(describe las reglas de formación de las sentencias).

Una Gramática libre de contexto se define como una $4-tupla\ G = \{S, \sum, N, P\}$, donde

- S representa el símbolo inicial
- N es el conjunto finito de los símbolos no terminales
- P es el conjunto de reglas de producción
- $\sum$ es el conjunto finito de símbolos terminales

Entonces, a continuación se define la grámatica de Minilisp. Para ello haremos uso de la notación **EBNF**.

- S representa el simbolo inicial
- $\sum =$

$$\{+, -, *, /, \text{sub1}, \text{add1}, \text{sqrt}, \text{exp}\} \quad - \textbf{ operadores aritemticos}$$
$$\cup \{=, <, >, >=, <=, !=\} \quad - \textbf{ operadores boleanos}$$
$$\cup \{\text{not}, \text{if}, \text{fst}, \text{snd}, \text{let}, \text{letsec}, \text{head}, \text{tail}, \text{lambda}, \text{app}\} \quad - \textbf{ palabras reservadas}$$

- N =

$$\{< Arg >, < Expr >\}$$
$$\cup \{< Num >, < Digit >, < Int >\}$$
$$\cup \{< Bool >\}$$
$$\cup \{< String >, < Letra >, < Caracter >\}$$
$$\cup \{< Par >\}$$
$$\cup \{< List >, < ArgList >\}$$
$$\cup \{< App >\}$$

  Donde $< Arg >$ se encargará qué de la aridad de nuestros operadores.
  Esto es el ugar perfecto para agegar la aridad minima de cada operador, pueto que desde la construccion de ¡Args¿ podemos obtener ciertas restriciones. Pero dado que la implementacion esta a nuestra dispocision, entonces decidimos hacer algo nuevo.

- P, el conjunto de reglas de produccion se define de la siguinte manera.

Sintaxis Libre de Contexto de Minilisp en EBNF

```
<S> := <Expr>
<Expr> ::= <Int>
          | <Bool>
          | <String>
          | (+ <Arg>)
          | (- <Arg>)
          | (* <Arg>)
          | (/ <Arg>)
          | (< <Arg>)
          | (> <Arg>)
          | (= <Arg>)
          | (<= <Arg>)
          | (>= <Arg>)
          | (!= <Arg>)
          | (add1 <Arg> )
          | (sub1 <Arg> )
          | (++ <Expr> <Expr> )
          | (expt <Expr> <Arg>)
          | (sqrt <Arg>)
          | (not <Expr>)
          | (and <Arg>)
          | <Par>
          | (fst <Expr>)
          | (snd <Expr>)
          | (let <Plet> <Expr>)
          | (letsec <Plet> <Expr>)
          | (letrec <Plet> <Expr>)
          | (if <Expr> <Expr> <Expr>)
          | <List>
```

```
          | (head <Expr>)
          | (tail <Expr>)
          | (lambda <Plambda> <Expr>) -- esto de
          | <App>
          | (cond <Pcond> else <Expr>)

<Par> := ( <Expr> , <Expr> ) -- par ordenado

<List>:= [] | [<ArgList>]
<ArgList>:= <Expr>,<ArgList> |  <Expr>   -- Argumento de la lista

<Arg>:= <Expr> <Arg> | [] -- argumento para operadores varidicos

<App>:= ( <Expr> <Arg>) -- aplicacion

<Bool>:= #t | #f

<String>:= <Letra> {<Caracter>}
<Letra>:= a | b | ... | z | A | ... | Z |
<Caracter> := <Letra> | _ | <Digit> | 0

<Plet>:= (<String> <Expr>) |  (<String> <Expr>) <Plet>

<Plambda>:= <String> | <String> <Plambda>

<Pguarda>:= '[' <expr> <expr> ']'
<Pcond>:=  <Pguarda> | <Pguarda> <Pcond>

<Int> := <Num>
        | -<Num>

<Digit> := 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Num> := 0 | <Digit>{<Num>}
```

Ahora analicemos cada una de las expresiones:
Comenzamos con los simbolos no termiales que representan las constantes. $< Int >, < Bool >, < String >$. Con estas construcciones logramos tener el conjunto de los enteros, y para las variables podremos usar snakeCase o camelCase, y finalmente los booleanos pues simplemente es true o false.

Ahora, pasameos con los simbolos terminales $+, -, *, /, <, >, =, <=, >=, !=, add1, sub1, sqrt$ podemos notar que reciben una lista de argumentos que puede que este vacía, sin embargo las restricciones de esto se implementaran en el desugar.
Cabe aclarar que pues se espera que las expresiones sean numeros, pero esto realmente no nos importa acá, sino simplemente saber lo que reciben, asi como estructura.
Ahora bien, notemos que $<$ **Arg** $>$ es un secuencia de expresiones separadas con un espacio, o bien puede no tener nada, la construccion de este no terminal esta claramente abusando de la notación EBNF, puesto que no existea algo como vacío(en este caso lo representamos con la lista vacía).

ahora pasando con los restantes

1. ++ : este recibe dos expresiones, la primera debera corresponder al elemento que se quiere concatenar a la lista, y el la segunda expresión corresponde al de la lista
2. **expt** : este recibe 2 cosas, la primera reprsenta el exponente al que se quiere elevar, y lo demas son los argumentos a los que se le aplicará la operación.
3. **not** : simplemente recibe una expresion que es la que se va a negar.
4. **and** : recibe una lista de expresiones. Este operador será de utilidad para describir el comportamiento de los operadores logicos.
5. **fst** : simplemente recibe un par ordenado
6. **snd** : este es analogo al fst.
7. **let, letsec, letrec**: cada uno de estos puede recibe 2 argumentos, donde el primero representa una lista de tuplas de identificadores y sus respectivos valores, y el segundo es el curerpo donde se sustituye cada valor.
8. **if** : este recibe tres argumentos(en este caso no se plantea soporte para varidicos), el primero representa la condicion, el segundo es lo que se deulve si se cumple la condicion, de otro modo se devuelve el tercer argumento.
9. head, tail : estos reciben simplemente la lista para operar con ella.
10. **lambda** : este recibe dos argumentos, el primero $< Plambda >$ represeta los identificadores y el segundo es el cuerpo de la función.
11. **cond, else** : cond es como un if con la diferencia de que recibe varias condiciones, esperando que se cumpla alguna de estas(la primera que cea cierta devuelve un valor), que de no serlo así, entonces se evalua el cuerpo del else.
    Entrando en contexto con la estructura de este operador, notese que cada elemento de de $< Pcond >$ es una gurda con dos elementos, el primero es la condición, y el segundo es la expresion a devolver en caso de que la condicion sea cierta. luego $< Expr >$ es el valor que se devolvera si no se cumple ninguna de las condiciones.

los no terminales $< Par >, < List >, < App >$:

1. $<$ **Par** $>$: esto simplmente son dos expresiones entre parentesis separadas por una coma, tal como lo es.
2. $<$ **List** $>$ para este construimos $< ArgList >$ que representa los elementos de la lista. pero este tiene como argumento una expresión, asi que para considerar la lista vacía creamos $< List >$.
3. $<$ **App** $>$: la aplicación recibe dos parametros, el primero es una función, así que esto es una expresión, y el segundo deben ser expresiones que sustituyan los a los identificadores de la fucnión, por lo que el segundo argumento es ¡Arg¿. La aplicacion se ve como dos expresiones dentro de parentesis.

2.3 Sintaxis abstracta

la sintaxis abstracta se encarga de capturar la lógica y la jerarquia del programa, omitiendo los detalles superficiales propios de la sintaxis concreta, es decir, omite elementos considerados azúcar sintactica. Comunmente, se representa mediante un *árbol de sintaxis abstracta*(**ASA**).

- Un **árbol de sintaxis abstracta** es un árbol o ordenado $A = (N, E, r)$, donde:

- N es el conjunto finito de nodos etiquetados qué representan construcciones del lenguaje; las hojas corresponden a valores constantes.
- $E \subseteq N \times N$ es el conjunto de aristas dirigidas entre nodos.
- $r \in N$ es la raíz del arbol.

- **Representacion mediante reglas de inferencia**

  Los ASA se pueden representar mediante reglas de inferencia qué definen la construcción de los nodos del árbol a partir de sus subexpresiones.

A continuación se proporcionan las reglas de inferencias para nuestro Minilisp, las cuales se han clasificado en dos tipos:

1. Atomicas: son aquellas que representan valores constantes.
2. Subexpresiones: son aquellas pueden tener como argumento otras subexpresiones.

Atomicos

Sintaxis abstracta

Al recordar, las cadenas permitidas para los identificadores son String $= \{s \mid s \in [a-z, A-Z][a-z, A-Z, \_, 0-9]^*\}$

$$\frac{i \in \text{String}}{\text{Id}(i)\ \text{ASA}}$$

dado que nuestra definición ya maneja perfectamente los número enteros, entonces,

$$\frac{n \in \mathbb{Z}}{\text{Num}(n)\ \text{ASA}}$$

finalmente, para los boleanos, el conjunto de valores esta definido por $Bool = \{\#t, \#f\}$

$$\frac{b \in Bool}{Boolean(b)\ ASA}$$

.

Subexpresiones

Ahora, pasaremos con las subexpresiones, sin embargo, dado que estamos manejando operadores varidicos, entonces el agumento que le pasamos representa una lista de expresiones, en este caso son los **ASA**.

Sintaxis abstracta

Operadores aritméticos varidicos
Para los operadores de suma, resta, division, multiplicacion, raíz y exponente, se implementa de la siguiente manera.

- **Suma**

$$\frac{a\ [\text{ASA}]}{\text{Add}(a)\ \text{ASA}}$$

- **Resta**

$$\frac{a\ [\text{ASA}]}{\text{Sub}(a)\ \text{ASA}}$$

- **Multiplicación**

$$\frac{a\ [\text{ASA}]}{\text{Mult}(a)\ \text{ASA}}$$

- **Division**

$$\frac{a\ [\text{ASA}]}{\text{Div}(a)\ \text{ASA}}$$

- **Incremento**

$$\frac{a\ [\text{ASA}]}{\text{Add1}(a)\ \text{ASA}}$$

- **Decremento**

$$\frac{a\ [\text{ASA}]}{\text{Sub1}(a)\ \text{ASA}}$$

- **Raiz**

$$\frac{a\ [\text{ASA}]}{\text{Sqrt}(a)\ \text{ASA}}$$

- **Exponente**

$$\frac{a\ \text{ASA} \quad b\ [\text{ASA}]}{\text{Expt}(a, b)\ \text{ASA}}$$

Pares ordenados

- **Par**

$$\frac{a\ \text{ASA} \quad b\ ASA}{\text{Pair}(a, b)\ \text{ASA}}$$

- **Fst**

$$\frac{p\ \text{ASA}}{\text{Fst}(p)\ \text{ASA}}$$

- **Snd**

$$\frac{p\ \text{ASA}}{\text{snd}(p)\ \text{ASA}}$$

Comparadores

- **Igual**

$$\frac{a\ [\text{ASA}]}{\text{Eq}(a)\ \text{ASA}}$$

- **Menor**

$$\frac{a\ [\text{ASA}]}{\text{Lt}(a)\ \text{ASA}}$$

- **Mayor**

$$\frac{a\ [\text{ASA}]}{\text{Gt}(a)\ \text{ASA}}$$

- **Menor o igual**

$$\frac{a\ [\text{ASA}]}{\text{Leq}(a)\ \text{ASA}}$$

- **Mayor o igual**

$$\frac{a\ [\text{ASA}]}{\text{Geq}(a)\ \text{ASA}}$$

- **Distinto**

$$\frac{a\ [\text{ASA}]}{\text{Neq}(a)\ \text{ASA}}$$

Operadores Lógicos

- **And**

$$\frac{a\ \text{ASA} \quad b\ \text{ASA}}{\text{And}(a,b)\ \text{ASA}}$$

- **Not**

$$\frac{b\ \text{ASA}}{\text{Not}(b)\ \text{ASA}}$$

- **If**

$$\frac{c\ \text{ASA} \quad t\ \text{ASA} \quad f\ \text{ASA}}{\text{If}(c,t,f)\ \text{ASA}}$$

Función Let y Let*

- **Let**

$$\frac{(i,v)\ [(String, \text{ASA})] \quad c\ \text{ASA}}{\text{Let}((i,v),c)\ \ ASA}$$

- **Let***

$$\frac{(i,v)\ [(String, \text{ASA})] \quad c\ \text{ASA}}{\text{Let}*((i,v),c)\ \ ASA}$$

- **Letrec**

$$\frac{(i,v)\ [(String, \text{ASA})] \quad c\ \text{ASA}}{\text{Letrec}((i,v),c)\ \ ASA}$$

Listas

- **List**

$$\frac{l\ [\text{ASA}]}{\text{List}(l)\ \text{ASA}}$$

- **Conc**

$$\frac{e\ \text{ASA} \quad l\ \text{ASA}}{\text{Conc}(e,l)\ \ \text{ASA}}$$

Condicional

- **Condicional Else**

$$\frac{(b,a) \in [(ASA, \text{ASA})]}{\text{CondElse}((b,a)) \quad ASA}$$

App y Lambda

- **Lambda**

$$\frac{i \in [String] \quad c \text{ ASA}}{\text{lambda}(i,c) \quad \text{ASA}}$$

- **App**

$$\frac{f \text{ ASA} \quad a \text{ ASA}}{\text{App}(f,a) \quad \text{ASA}}$$

2.4 Desugar

Se le conoce a **azúcar sintáctica** a la construcciones qué puedan expresarse haciendo uso de otras construcciones del lenguaje, es decir, son las construcciones qué se puedan traducir a otras construcciones más fundamentales. Esto es de suma importancia, pues lo que se pretende es pasar de la superficie(flujo de texto recibido) al nucleo(se llevan acabo las construcciones mínimas qué resuleven las construcciones de la superficie). Para Minilisp es muy importante manejar en caso de los operadores varidos, puesto que se pretende que en el núcleo solo se reciban los elementos necesarios.

Un ejemplo de una construcción desasucarizada es la función **let**.
$(let(x \;\; 2)(+ \;\; 2 \;\; x))$ es equivalente a $(App(lambda(x)(+ \;\; 2 \;\; x)))$.

- Comenzaremos por definir un tipo de dato qué represente a las expresiones desasucarizadas. A este tipo de dato lo llamaremos **ASAValues**. Este tiene cierto parecido con la construcción de los ASA con la diferencia de que acá ya se estará trabajando con el nucleo, es decir, se pretende que las operaciones se realicen con las unidades mínimas necesarias.

  A continuación se muestra la estructura de cada operación en el núcleo.

```
data ASAValues
 = IdV String      -- las identificadores son cadenas
 | NumV Int        -- los numeros son enteros
 | BooleanV Bool   -- los Booleanos
    -- ==================
    -- aritméticas (suma, resta, multiplicacion, division) -> (AddV, SubV, MultV, DivV)
    -- ==================
 | Op ASAValues ASAValues        -- Cada uno recibe dos argumentos

    -- ==================
    -- logicos
    -- ==================
 | AndV ASAValues ASAValues                -- el and recibe dos argumentos
 | NotV ASAValues                          -- el not recibe un solo argumento
 | IfV ASAValues ASAValues ASAValues       -- el if recibe 3 argumentos
```

```
    -- ==================
    -- comparadores (<=, >= , <, >, =, != ) -> (LetV, GeqV, Lt, Gt, EqV, NeqV)
    -- ==================
  | Op ASAValues ASAValues                  -- cada uno de estos dos tres argumentos

    -- ==================
    -- pares ordenados
    -- ==================
  | PairV ASAValues ASAValues                -- Cada par ordenado recibe dos argumentos
  | FstV ASAValues                           -- la función Firts recibe un solo argumento
  | SndV ASAValues                           -- la función Secodn recibe un solo argumento

    -- ==================
    -- listas
    -- ==================
  | NilV                                     -- El núcleo representa
  | ConsV ASAValues ASAValues                --
  | HeadV ASAValues                          -- la función head recibe un solo argumento
  | TailV ASAValues                          -- la función tail recibe un solo argumento
    --cosas con let
  | FunV String ASAValues   -- la función lambda recibe dos argumentos(identificador y cuerpo)

  | ClosureV String ASAValues [(String, ASAValues)]

  | AppV ASAValues ASAValues   -- Una aplicación recibe dos argumentos(función y valor)
 deriving (Show)
```

Nótese qué se está implementando otros tipos de construcciones qué no existían en la sintaxis, estas son **ClosureV, NilV y ConsV** . Así que a continuación se explica el porqué de su implementación.

- **ClousureV:** Esta construcción representa una cerradura que será de utilidad para la semántica operacional, puesto que se implementará alcance estático. Las partes que conforman la cerradura son las que la definen:

  1. el parámetro de la función
  2. el cuerpo de la función
  3. el ambiente (este se puede representar como una lista de tuplas de un identificador un una construcción del nucleo(ASAValues))

  la implementación de ClousureV aquí se debe a que justamente la semantica operación solo con elementos del nucleo.

- **Nilv:** Este representa la lista vacía [], y también el tope de la lista.

- **ConsV:** Esta costrucción es el contenedor de para los elementos de la lista

3 Desazucarización de Operadores Variádicos

La extensión de MiniLisp con operadores variádicos requirió de desazucarización hacia el núcleo del lenguaje. Aquí se presenta la formalización de este proceso, distinguiendo tres categorías de operadores según su aridad y comportamiento.

3.1 Clasificación de Operadores

3.1.1 Operadores Unarios Generalizados

Los operadores unarios generalizados son aquellos que se aplican elemento por elemento sobre una lista de argumentos, creada por el archivo *Grammars.y*, produciendo una lista de resultados. Ejemplos de estos operadores son `add1`, `sub1` y `sqrt`.

Regla de desazucarización:

Sea $\oplus$ un operador unario y $[e_1, e_2, \ldots, e_n]$ una lista de expresionesn ASA (no del núcleo). La desazucarización se define inductivamente como:

$$\begin{aligned} \oplus\ [] &= \texttt{NilV} \\ \oplus\ [e_1] &= \texttt{ConsV}\ (\oplus\ e_1)\ \texttt{NilV} \\ \oplus\ (e_1 : \text{resto}) &= \texttt{ConsV}\ (\oplus\ e_1)\ (\oplus\ \text{resto}) \end{aligned}$$

Ejemplo: La expresión `(add1 1 2 3)` se desazucariza como:

```
ConsV (AddV (NumV 1) (NumV 1)) (ConsV (AddV (NumV 2) (NumV 1)) (ConsV (AddV (NumV 3) (NumV 1)) NilV))
```

Esta transformación preserva el orden de evaluación de izquierda a derecha y mantiene la estructura de lista requerida para operaciones que procesan múltiples valores de manera uniforme.

Cabe aclarar en el caso en el que los operadores reciben 1 argumento(o los argumentos minimos), la regla de desucarizacion se remplaza por la forma normal, un ejemplo de ello es desugar (Add1 x) = AddV (desugar x) (NumV 1). Esto proporciona la estructura correcta para poder seguir aplicando operaciones que requieran de valores constantes. en sintaxis concreta, yo podria operar con (+ 1 (add1 2)), pero no con (+ 1 (add1 2 3)), puesto que el primer add1 me devuelve un valor, pero el segundo me devuelve una lista.

3.1.2 Operadores Binarios Asociativos

Los operadores binarios (como `+`, `-`, `*`, `/`) requieren al menos dos argumentos y se desazucarizan mediante asociatividad.

Reglas de desazucarización:

Sea $\odot$ un operador binario. La desazucarización se define como:

$$\begin{aligned} \odot\ [e_1, e_2] &= \odot\ e_1\ e_2 \\ \odot\ [e_1, e_2, \ldots, e_n] &= \odot\ e_1\ (\odot\ [e_2, \ldots, e_n]) \quad \text{para } n > 2 \end{aligned}$$

Ejemplo: La expresión `(+ 1 2 3 4)` se desazucariza como:

```
AddV (NumV 1) (AddV (NumV 2) (AddV (NumV 3) (NumV 4)))
```

Caso especial - Resta unaria: La expresión `(- x)` se interpreta como negación aritmética y se desazucariza como:

(A partir de aquí el lado izquierda de las igualdades se escribirán con sintaxis de código para mejor visualización aunque deberían ser ASAs pues la función va de ASA a ASAValues)

$$\texttt{(- } e\texttt{)} = \texttt{(* -1 } e\texttt{)} = \texttt{MultV (NumV -1) } e$$

3.1.3 Operadores de Comparación Variádicos

Los operadores de comparación (`=`, `<`, `>`, `<=`, `>=`, `!=`) con más de dos argumentos se interpretan como una cadena de comparaciones que deben cumplirse simultáneamente de acuerdo a la especificación que se definió.

Regla de desazucarización:
Sea $\sim$ un operador de comparación. Para una lista de al menos tres elementos, la desazucarización es:

$$\begin{aligned} \sim\ [e_1, e_2] &= \sim\ e_1\ e_2 \\ \sim\ [e_1, e_2, e_3, \ldots, e_n] &= \texttt{AndV } (\sim\ e_1\ e_2)\ (\sim\ [e_2, e_3, \ldots, e_n]) \end{aligned}$$

Ejemplo: La expresión `(< 1 2 3 4)` se desazucariza como:

```
AndV (LtV (NumV 1) (NumV 2)) (AndV (LtV (NumV 2) (NumV 3)) (LtV (NumV 3) (NumV 4)))
```

Esto expresa la condición: $1 < 2 \land 2 < 3 \land 3 < 4$.
La motivación para esta regla es que cada par adyacente debe satisfacer la relación.

3.2 Observaciones Adicionales sobre Desazucarización

3.2.1 Exponenciación Variádica

El operador `expt` presenta un caso especial, ya que acepta una base fija y múltiples exponentes:

$$\begin{aligned} \texttt{(expt } b\ [e]\texttt{)} &= \texttt{ExpV } b\ e \\ \texttt{(expt } b\ [e_1, \ldots, e_n]\texttt{)} &= \texttt{ConsV (ExpV } e_1\ b\texttt{) (expt } b\ [e_2, \ldots, e_n]\texttt{)} \end{aligned}$$

3.2.2 Curryficación de Funciones

Todas las funciones en el núcleo se expresan en forma currificada. Una función lambda con múltiples parámetros:

$$\texttt{(lambda (}x_1\ x_2\ \ldots\ x_n\texttt{) } e\texttt{)}$$

se desazucariza como una cadena de funciones unarias anidadas:

$$\texttt{(lambda (}x_1\ x_2\ \ldots\ x_n\texttt{) } e\texttt{)} = \texttt{FunV } x_1\ (\texttt{FunV } x_2\ (\ldots\ (\texttt{FunV } x_n\ e)))$$

La aplicación de funciones se desazucariza correspondientemente:

$$\texttt{(}f\ e_1\ e_2\ \ldots\ e_n\texttt{)} = \texttt{AppV } (\ldots\ (\texttt{AppV (AppV } f\ e_1\texttt{) } e_2\texttt{)}\ \ldots)\ e_n$$

3.2.3 Let y Let*

El operador `let` con múltiples bindings se desazucariza como una aplicación de función:

$$\texttt{(let ((}x_1\ e_1\texttt{)} \ldots \texttt{(}x_n\ e_n\texttt{)) } b\texttt{)} = \texttt{AppV (FunV } x_1\ (\ldots\ (\texttt{FunV } x_n\ b)\ \ldots))\ e_1\ \ldots\ e_n$$

El operador `let*` se desazucariza secuencialmente:

$$\begin{aligned} \texttt{(let* ((}x\ e\texttt{)) } b\texttt{)} &= \texttt{AppV (FunV } x\ b\texttt{) } e \\ \texttt{(let* ((}x_1\ e_1\texttt{)} \ldots \texttt{(}x_n\ e_n\texttt{)) } b\texttt{)} &= \texttt{AppV (FunV } x_1\ \texttt{(let* ((}x_2\ e_2\texttt{)} \ldots\texttt{)}b\texttt{))}\ e_1 \end{aligned}$$

3.2.4 Condicional Cond

La construcción `cond` se desazucariza recursivamente a `if`:

$$\texttt{(cond [}g_1\ e_1\texttt{] [else }e_n\texttt{])} = \texttt{IfV}\ g_1\ e_1\ e_n$$
$$\texttt{(cond [}g_1\ e_1\texttt{]}\ldots\texttt{[}g_n\ e_n\texttt{] [else }e_{else}\texttt{])} = \texttt{IfV}\ g_1\ e_1\ \texttt{((cond [}g_2\ e_2\texttt{]}\ldots\texttt{[else }e_{else}\texttt{]))}$$

3.2.5 Listas

La notación con corchetes para listas se desazucariza a constructores `ConsV` y `NilV`:

$$\texttt{[]} = \texttt{NilV}$$
$$\texttt{[}e_1, \ldots, e_n\texttt{]} = \texttt{ConsV}\ e_1\ (\texttt{[}e_2, \ldots, e_n\texttt{]})$$

3.2.6 Conc

La construcción concatenar "++" se desasucariza de la siguiente mmanera:

```
desugar (Conc e l) = ConcV (desugar e) (desugar l)
```

Otra posible solución es verlo directamente como azúcar sintáctica, es decir como desugar (ConsV e l), sin embargo, la desventaja que tiene es que e puede llegar a ser NilV, es decir, la lista vacía [], por lo que al concatenar quedaría la lista [[],....], que no es algo muy agradable para el usuario, además de ser ineficiente.

4 Semántica Operacional Estructural

A continuación se presenta la formalización completa de la semántica operacional de MiniLisp mediante reglas de inferencia en estilo de paso pequeño (small-step semantics). Los juicios de transición tienen la forma $\langle e, \varepsilon\rangle \to \langle e', \varepsilon'\rangle$, donde e es una expresión, ε es el ambiente y la flecha indica una transición en un paso.

4.1 Valores y Constantes

Reglas de traducción en paso pequeño

Axiomas para Valores
Constantes numéricas:

$$\frac{}{\langle \texttt{NumV}(n), \varepsilon\rangle \to \langle \texttt{NumV}(n), \varepsilon\rangle} \quad \text{[Num-Value]}$$

Constantes booleanas:

$$\frac{}{\langle \texttt{BooleanV}(b), \varepsilon\rangle \to \langle \texttt{BooleanV}(b), \varepsilon\rangle} \quad \text{[Bool-Value]}$$

Lista vacía:

$$\frac{}{\langle \texttt{NilV}, \varepsilon\rangle \to \langle \texttt{NilV}, \varepsilon\rangle} \quad \text{[Nil-Value]}$$

Clausuras:

$$\frac{}{\langle \texttt{ClosureV}(p, c, \varepsilon'), \varepsilon\rangle \to \langle \texttt{ClosureV}(p, c, \varepsilon'), \varepsilon\rangle} \quad \text{[Closure-Value]}$$

4.2 Variables y Ambiente

Reglas de traducción en paso pequeño

Resolución de Variables

$$\frac{\text{lookup}(x, \varepsilon) = v}{\langle \texttt{IdV}(x), \varepsilon \rangle \to \langle v, \varepsilon \rangle} \quad \text{[Var-Lookup]}$$

$$\frac{\text{lookup}(x, \varepsilon) \text{ no encontrado (error)}}{\langle \texttt{IdV}(x), \varepsilon \rangle \to \text{error}} \quad \text{[Var-Error]}$$

Definición de lookup: Es la función auxiliar $\text{lookup}(x, \varepsilon)$ busca la variable x en el ambiente ε:

$$\text{lookup}(x, []) = \text{error}$$

$$\text{lookup}(x, (y, v) :: \varepsilon') = \begin{cases} v & \text{si } x = y \\ \text{lookup}(x, \varepsilon') & \text{si } x \neq y \end{cases}$$

4.3 Operadores Aritméticos

Reglas de traducción en paso pequeño

Suma

Reducir operando izquierdo:

$$\frac{\langle i, \varepsilon \rangle \to \langle i', \varepsilon' \rangle}{\langle \texttt{AddV}(i, d), \varepsilon \rangle \to \langle \texttt{AddV}(i', d), \varepsilon \rangle} \quad \text{[Add-Left]}$$

Reducir operando derecho:

$$\frac{\langle d, \varepsilon \rangle \to \langle d', \varepsilon' \rangle}{\langle \texttt{AddV}(\texttt{NumV}(n_1), d), \varepsilon \rangle \to \langle \texttt{AddV}(\texttt{NumV}(n_1), d'), \varepsilon \rangle} \quad \text{[Add-Right]}$$

Reducir suma:

$$\frac{}{\langle \texttt{AddV}(\texttt{NumV}(n_1), \texttt{NumV}(n_2)), \varepsilon \rangle \to \langle \texttt{NumV}(n_1 +_{\mathbb{Z}} n_2), \varepsilon \rangle} \quad \text{[Redcue-sum]}$$

Reglas de traducción en paso pequeño

Resta

Reducir operando izquierdo:

$$\frac{\langle i, \varepsilon \rangle \to \langle i', \varepsilon' \rangle}{\langle \texttt{SubV}(i, d), \varepsilon \rangle \to \langle \texttt{SubV}(i', d), \varepsilon \rangle} \quad \text{[Sub-Left]}$$

Reducir operando derecho:

$$\frac{\langle d, \varepsilon \rangle \to \langle d', \varepsilon' \rangle}{\langle \texttt{SubV}(\texttt{NumV}(n_1), d), \varepsilon \rangle \to \langle \texttt{SubV}(\texttt{NumV}(n_1), d'), \varepsilon \rangle} \quad \text{[Sub-Right]}$$

Reducir resta:

$$\frac{}{\langle \texttt{SubV}(\texttt{NumV}(n_1), \texttt{NumV}(n_2)), \varepsilon \rangle \to \langle \texttt{NumV}(n_1 -_{\mathbb{Z}} n_2), \varepsilon \rangle} \quad \text{[Reduce-sub]}$$

Reglas de traducción en paso pequeño

Multiplicación

Reducir operando izquierdo:

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon'\rangle}{\langle \texttt{MultV}(i, d), \varepsilon\rangle \to \langle \texttt{MultV}(i', d), \varepsilon\rangle} \quad \text{[Mult-Left]}$$

Reducir operando derecho:

$$\frac{\langle d, \varepsilon\rangle \to \langle d', \varepsilon'\rangle}{\langle \texttt{MultV}(\texttt{NumV}(n_1), d), \varepsilon\rangle \to \langle \texttt{MultV}(\texttt{NumV}(n_1), d'), \varepsilon\rangle} \quad \text{[Mult-Right]}$$

Reducir multiplicación:

$$\frac{}{\langle \texttt{MultV}(\texttt{NumV}(n_1), \texttt{NumV}(n_2)), \varepsilon\rangle \to \langle \texttt{NumV}(n_1 \times_{\mathbb{Z}} n_2), \varepsilon\rangle} \quad \text{[Reduce-Mult]}$$

Reglas de traducción en paso pequeño

División

Reducir operando izquierdo:

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon'\rangle}{\langle \texttt{DivV}(i, d), \varepsilon\rangle \to \langle \texttt{DivV}(i', d), \varepsilon\rangle} \quad \text{[Div-Left]}$$

Reducir operando derecho:

$$\frac{\langle d, \varepsilon\rangle \to \langle d', \varepsilon'\rangle}{\langle \texttt{DivV}(\texttt{NumV}(n_1), d), \varepsilon\rangle \to \langle \texttt{DivV}(\texttt{NumV}(n_1), d'), \varepsilon\rangle} \quad \text{[Div-Right]}$$

Error de división por cero:

$$\frac{}{\langle \texttt{DivV}(\texttt{NumV}(n_1), \texttt{NumV}(0)), \varepsilon\rangle \to \text{error}} \quad \text{[Div-Zero-Error]}$$

Reducir división:

$$\frac{n_2 \neq 0}{\langle \texttt{DivV}(\texttt{NumV}(n_1), \texttt{NumV}(n_2)), \varepsilon\rangle \to \langle \texttt{NumV}(n_1 \div_{\mathbb{Z}} n_2), \varepsilon\rangle} \quad \text{[Reducir-Div]}$$

Reglas de traducción en paso pequeño

Exponenciación

Reducir base:

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon'\rangle}{\langle \texttt{ExpV}(i, d), \varepsilon\rangle \to \langle \texttt{ExpV}(i', d), \varepsilon\rangle} \quad \text{[Exp-Left]}$$

Reducir exponente:

$$\frac{\langle d, \varepsilon\rangle \to \langle d', \varepsilon'\rangle}{\langle \texttt{ExpV}(\texttt{NumV}(n_1), d), \varepsilon\rangle \to \langle \texttt{ExpV}(\texttt{NumV}(n_1), d'), \varepsilon\rangle} \quad \text{[Exp-Right]}$$

Reducir potencia:

$$\frac{}{\langle \texttt{ExpV}(\texttt{NumV}(n_1), \texttt{NumV}(n_2)), \varepsilon\rangle \to \langle \texttt{NumV}(n_1^{n_2}), \varepsilon\rangle} \quad \text{[Reducir-Exp]}$$

Reglas de traducción en paso pequeño

Raíz Cuadrada

Reducir argumento:

$$\frac{\langle e, \varepsilon\rangle \to \langle e', \varepsilon'\rangle}{\langle \texttt{SqrV}(e), \varepsilon\rangle \to \langle \texttt{SqrV}(e'), \varepsilon\rangle} \quad \text{[Sqrt-Reduce]}$$

Error para números negativos:

$$\frac{n < 0}{\langle \texttt{SqrV}(\texttt{NumV}(n)), \varepsilon\rangle \to \text{error}} \quad \text{[Sqrt-Negative-Error]}$$

Reducir raíz:

$$\frac{n \geq 0}{\langle \texttt{SqrV}(\texttt{NumV}(n)), \varepsilon\rangle \to \langle \texttt{NumV}(\lfloor\sqrt{n}\rfloor), \varepsilon\rangle} \quad \text{[Reduce-Sqrt]}$$

4.4 Operadores de Comparación

Reglas de traducción en paso pequeño

Igualdad

Comparación de listas vacías:

$$\frac{}{\langle \texttt{EqV}(\texttt{NilV}, \texttt{NilV}), \varepsilon\rangle \to \langle \texttt{BooleanV}(\text{True}), \varepsilon\rangle} \quad \text{[Eq-Nil]}$$

Reducir operando izquierdo:

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon'\rangle}{\langle \texttt{EqV}(i, d), \varepsilon\rangle \to \langle \texttt{EqV}(i', d), \varepsilon\rangle} \quad \text{[Eq-Left]}$$

Reducir operando derecho:

$$\frac{\langle d, \varepsilon\rangle \to \langle d', \varepsilon'\rangle}{\langle \texttt{EqV}(\texttt{NumV}(n), d), \varepsilon\rangle \to \langle \texttt{EqV}(\texttt{NumV}(n), d'), \varepsilon\rangle} \quad \text{[Eq-Right]}$$

Reducir igualdad:

$$\frac{}{\langle \texttt{EqV}(\texttt{NumV}(n_1), \texttt{NumV}(n_2)), \varepsilon\rangle \to \langle \texttt{BooleanV}(n_1 = n_2), \varepsilon\rangle} \quad \text{[ReduceEq]}$$

Reglas de traducción en paso pequeño

Menor Que

Reducir operando izquierdo:

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon'\rangle}{\langle \texttt{LtV}(i, d), \varepsilon\rangle \to \langle \texttt{LtV}(i', d), \varepsilon\rangle} \quad \text{[Lt-Left]}$$

Reducir operando derecho:

$$\frac{\langle d, \varepsilon\rangle \to \langle d', \varepsilon'\rangle}{\langle \texttt{LtV}(\texttt{NumV}(n), d), \varepsilon\rangle \to \langle \texttt{LtV}(\texttt{NumV}(n), d'), \varepsilon\rangle} \quad \text{[Lt-Right]}$$

Computar comparación:

$$\frac{}{\langle \texttt{LtV}(\texttt{NumV}(n_1), \texttt{NumV}(n_2)), \varepsilon\rangle \to \langle \texttt{BooleanV}(n_1 < n_2), \varepsilon\rangle} \quad \text{[Lt-Compute]}$$

Reglas de traducción en paso pequeño

Mayor Que

Reducir operando izquierdo:

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon'\rangle}{\langle \texttt{GtV}(i, d), \varepsilon\rangle \to \langle \texttt{GtV}(i', d), \varepsilon\rangle} \quad \text{[Gt-Left]}$$

Reducir operando derecho:

$$\frac{\langle d, \varepsilon\rangle \to \langle d', \varepsilon'\rangle}{\langle \texttt{GtV}(\texttt{NumV}(n), d), \varepsilon\rangle \to \langle \texttt{GtV}(\texttt{NumV}(n), d'), \varepsilon\rangle} \quad \text{[Gt-Right]}$$

Reducir comparación:

$$\frac{}{\langle \texttt{GtV}(\texttt{NumV}(n_1), \texttt{NumV}(n_2)), \varepsilon\rangle \to \langle \texttt{BooleanV}(n_1 > n_2), \varepsilon\rangle} \quad \text{[Redcue-Gt]}$$

Reglas de traducción en paso pequeño

Menor o Igual Que

Reducir operando izquierdo:

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon'\rangle}{\langle \texttt{LeqV}(i, d), \varepsilon\rangle \to \langle \texttt{LeqV}(i', d), \varepsilon\rangle} \quad \text{[Leq-Left]}$$

Reducir operando derecho:

$$\frac{\langle d, \varepsilon\rangle \to \langle d', \varepsilon'\rangle}{\langle \texttt{LeqV}(\texttt{NumV}(n), d), \varepsilon\rangle \to \langle \texttt{LeqV}(\texttt{NumV}(n), d'), \varepsilon\rangle} \quad \text{[Leq-Right]}$$

Reducir comparación:

$$\frac{}{\langle \texttt{LeqV}(\texttt{NumV}(n_1), \texttt{NumV}(n_2)), \varepsilon\rangle \to \langle \texttt{BooleanV}(n_1 \leq n_2), \varepsilon\rangle} \quad \text{[Reduce-Leq]}$$

Reglas de traducción en paso pequeño

Mayor o Igual Que

Reducir operando izquierdo:

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon'\rangle}{\langle \texttt{GeqV}(i, d), \varepsilon\rangle \to \langle \texttt{GeqV}(i', d), \varepsilon\rangle} \quad \text{[Geq-Left]}$$

Reducir operando derecho:

$$\frac{\langle d, \varepsilon\rangle \to \langle d', \varepsilon'\rangle}{\langle \texttt{GeqV}(\texttt{NumV}(n), d), \varepsilon\rangle \to \langle \texttt{GeqV}(\texttt{NumV}(n), d'), \varepsilon\rangle} \quad \text{[Geq-Right]}$$

Reducir comparación:

$$\frac{}{\langle \texttt{GeqV}(\texttt{NumV}(n_1), \texttt{NumV}(n_2)), \varepsilon\rangle \to \langle \texttt{BooleanV}(n_1 \geq n_2), \varepsilon\rangle} \quad \text{[Reduce-Geq]}$$

Reglas de traducción en paso pequeño

Desigualdad

Reducir operando izquierdo:

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon'\rangle}{\langle \texttt{NeqV}(i, d), \varepsilon\rangle \to \langle \texttt{NeqV}(i', d), \varepsilon\rangle} \quad \text{[Neq-Left]}$$

Reducir operando derecho:

$$\frac{\langle d, \varepsilon\rangle \to \langle d', \varepsilon'\rangle}{\langle \texttt{NeqV}(\texttt{NumV}(n), d), \varepsilon\rangle \to \langle \texttt{NeqV}(\texttt{NumV}(n), d'), \varepsilon\rangle} \quad \text{[Neq-Right]}$$

Reducir desigualdad:

$$\frac{}{\langle \texttt{NeqV}(\texttt{NumV}(n_1), \texttt{NumV}(n_2)), \varepsilon\rangle \to \langle \texttt{BooleanV}(n_1 \neq n_2), \varepsilon\rangle} \quad \text{[Reduce-Neq]}$$

4.5 Operadores Lógicos

Reglas de traducción en paso pequeño

Negación

Reducir argumento:

$$\frac{\langle e, \varepsilon\rangle \to \langle e', \varepsilon'\rangle}{\langle \texttt{NotV}(e), \varepsilon\rangle \to \langle \texttt{NotV}(e'), \varepsilon\rangle} \quad \text{[Not-Reduce]}$$

Reducir negación:

$$\frac{}{\langle \texttt{NotV}(\texttt{BooleanV}(b)), \varepsilon\rangle \to \langle \texttt{BooleanV}(\neg b), \varepsilon\rangle} \quad \text{[Reduce-Not]}$$

Reglas de traducción en paso pequeño

Conjunción

Reducir operando izquierdo:

$$\frac{\langle e_1, \varepsilon\rangle \to \langle e_1', \varepsilon'\rangle}{\langle \texttt{AndV}(e_1, e_2), \varepsilon\rangle \to \langle \texttt{AndV}(e_1', e_2), \varepsilon\rangle} \quad \text{[And-Left]}$$

Reducir operando derecho:

$$\frac{\langle e_2, \varepsilon\rangle \to \langle e_2', \varepsilon'\rangle}{\langle \texttt{AndV}(\texttt{BooleanV}(b), e_2), \varepsilon\rangle \to \langle \texttt{AndV}(\texttt{BooleanV}(b), e_2'), \varepsilon\rangle} \quad \text{[And-Right]}$$

Reducir conjunción:

$$\frac{}{\langle \texttt{AndV}(\texttt{BooleanV}(b_1), \texttt{BooleanV}(b_2)), \varepsilon\rangle \to \langle \texttt{BooleanV}(b_1 \wedge b_2), \varepsilon\rangle} \quad \text{[Reduce-And]}$$

4.6 Condicionales

Reglas de traducción en paso pequeño

If
Reducir condición:

$$\frac{\langle e_1, \varepsilon\rangle \to \langle e_1', \varepsilon'\rangle}{\langle \texttt{IfV}(e_1, e_2, e_3), \varepsilon\rangle \to \langle \texttt{IfV}(e_1', e_2, e_3), \varepsilon'\rangle} \quad \text{[If-Cond]}$$

Rama verdadera:

$$\frac{}{\langle \texttt{IfV}(\texttt{BooleanV}(\text{True}), e_2, e_3), \varepsilon\rangle \to \langle e_2, \varepsilon\rangle} \quad \text{[If-True]}$$

Rama falsa:

$$\frac{}{\langle \texttt{IfV}(\texttt{BooleanV}(\text{False}), e_2, e_3), \varepsilon\rangle \to \langle e_3, \varepsilon\rangle} \quad \text{[If-False]}$$

4.7 Pares Ordenados

Reglas de traducción en paso pequeño

Construcción y Proyección de Pares
Reducir primer componente:

$$\frac{\langle e_1, \varepsilon\rangle \to \langle e_1', \varepsilon'\rangle}{\langle \texttt{PairV}(e_1, e_2), \varepsilon\rangle \to \langle \texttt{PairV}(e_1', e_2), \varepsilon\rangle} \quad \text{[Pair-Left]}$$

Reducir segundo componente:

$$\frac{\langle e_2, \varepsilon\rangle \to \langle e_2', \varepsilon'\rangle \quad \text{isValue}(v_1)}{\langle \texttt{PairV}(v_1, e_2), \varepsilon\rangle \to \langle \texttt{PairV}(v_1, e_2'), \varepsilon\rangle} \quad \text{[Pair-Right]}$$

Primera proyección - reducir argumento:

$$\frac{\langle e, \varepsilon\rangle \to \langle e', \varepsilon'\rangle}{\langle \texttt{FstV}(e), \varepsilon\rangle \to \langle \texttt{FstV}(e'), \varepsilon\rangle} \quad \text{[Fst-Reduce]}$$

Primera proyección - extraer:

$$\frac{\text{isValue}(v_1) \quad \text{isValue}(v_2)}{\langle \texttt{FstV}(\texttt{PairV}(v_1, v_2)), \varepsilon\rangle \to \langle v_1, \varepsilon\rangle} \quad \text{[Fst-Extract]}$$

Segunda proyección - reducir argumento:

$$\frac{\langle e, \varepsilon\rangle \to \langle e', \varepsilon'\rangle}{\langle \texttt{SndV}(e), \varepsilon\rangle \to \langle \texttt{SndV}(e'), \varepsilon\rangle} \quad \text{[Snd-Reduce]}$$

Segunda proyección - extraer:

$$\frac{\text{isValue}(v_1) \quad \text{isValue}(v_2)}{\langle \texttt{SndV}(\texttt{PairV}(v_1, v_2)), \varepsilon\rangle \to \langle v_2, \varepsilon\rangle} \quad \text{[Snd-Extract]}$$

4.8 Listas

Reglas de traducción en paso pequeño

Construcción de Listas

Cons con segundo argumento NilV - reducir primer elemento:

$$\frac{\langle e_1, \varepsilon\rangle \to \langle e_1', \varepsilon'\rangle \quad \neg\text{isValue}(e_1)}{\langle \texttt{ConsV}(e_1, \texttt{NilV}), \varepsilon\rangle \to \langle \texttt{ConsV}(e_1', \texttt{NilV}), \varepsilon\rangle} \quad \text{[Cons-Nil-Head]}$$

Cons con segundo argumento ConsV - reducir primer elemento:

$$\frac{\langle e_1, \varepsilon\rangle \to \langle e_1', \varepsilon'\rangle \quad \neg\text{isValue}(e_1)}{\langle \texttt{ConsV}(e_1, \texttt{ConsV}(h, t)), \varepsilon\rangle \to \langle \texttt{ConsV}(e_1', \texttt{ConsV}(h, t)), \varepsilon\rangle} \quad \text{[Cons-List-Head]}$$

Cons con primer elemento valor - reducir resto:

$$\frac{\langle e_2, \varepsilon\rangle \to \langle e_2', \varepsilon'\rangle \quad \text{isValue}(v_1)}{\langle \texttt{ConsV}(v_1, e_2), \varepsilon\rangle \to \langle \texttt{ConsV}(v_1, e_2'), \varepsilon\rangle} \quad \text{[Cons-Tail]}$$

Cons con argumento general:

$$\frac{\langle e_2, \varepsilon\rangle \to \langle e_2', \varepsilon'\rangle \quad e_2 \neq \texttt{NilV} \quad e_2 \neq \texttt{ConsV}(\ldots)}{\langle \texttt{ConsV}(e_1, e_2), \varepsilon\rangle \to \langle \texttt{ConsV}(e_1, e_2'), \varepsilon\rangle} \quad \text{[Cons-General]}$$

Reglas de traducción en paso pequeño

Construcción de Concatenar

ConcV de 3 casos especiales

$$\frac{}{\langle \texttt{ConcV}(\texttt{ConsV}(e_1\ e_2), \texttt{NilV}), \varepsilon\rangle \to \langle \texttt{ConsV}(e_1\ e_2), \varepsilon\rangle} \quad \text{[caso 1]}$$

$$\frac{}{\langle \texttt{ConcV}(\texttt{NilV}, \texttt{ConsV}(e_1\ e_2)-), \varepsilon\rangle \to \langle \texttt{ConsV}(e_1\ e_2), \varepsilon\rangle} \quad \text{[caso 2]}$$

$$\frac{}{\langle \texttt{ConcV}(\texttt{NilV}, \texttt{NilV}), \varepsilon\rangle \to \langle \texttt{NilV}, \varepsilon\rangle} \quad \text{[caso 3]}$$

Los dos son valores, pero ninguno de los anteriores

$$\frac{v_2 \neq \texttt{Cons}\ e_1\ e_2}{\langle \texttt{ConcV}(v_1, v_2), \varepsilon\rangle \to \textit{error} \quad \textit{" el segundo argumento debe ser una lista"}} \quad \text{[caso error]}$$

ConsV con el primer valor reducido

$$\frac{\langle l, \varepsilon\rangle \to \langle l', \varepsilon'\rangle}{\langle \texttt{ConcV}(v_1, l), \varepsilon\rangle \to \langle \texttt{ConcV}(v_1, l'), \varepsilon\rangle} \quad \text{[caso que sigue]}$$

ConsV con argumento general

$$\frac{\langle e, \varepsilon\rangle \to \langle e', \varepsilon'\rangle}{\langle \texttt{ConcV}(e, l), \varepsilon\rangle \to \langle \texttt{ConcV}(e', l), \varepsilon\rangle} \quad \text{[caso que sigue]}$$

Reglas de traducción en paso pequeño

Head de Lista

Error en lista vacía:

$$\frac{}{\langle \texttt{HeadV}(\texttt{NilV}), \varepsilon\rangle \to \text{error}} \quad \text{[Head-Nil-Error]}$$

Extraer cabeza (cuando es valor):

$$\frac{\text{isValue}(v)}{\langle \texttt{HeadV}(\texttt{ConsV}(v, rest)), \varepsilon\rangle \to \langle v, \varepsilon\rangle} \quad \text{[Head-Extract]}$$

Reducir lista cuando cabeza no es valor:

$$\frac{\langle \texttt{ConsV}(e, r), \varepsilon\rangle \to \langle \texttt{ConsV}(e', r'), \varepsilon\rangle \quad \neg\text{isValue}(e)}{\langle \texttt{HeadV}(\texttt{ConsV}(e, r)), \varepsilon\rangle \to \langle \texttt{HeadV}(\texttt{ConsV}(e', r')), \varepsilon\rangle} \quad \text{[Head-Reduce-List]}$$

Reducir expresión general:

$$\frac{\langle e, \varepsilon\rangle \to \langle e', \varepsilon\rangle}{\langle \texttt{HeadV}(e), \varepsilon\rangle \to \langle \texttt{HeadV}(e'), \varepsilon\rangle} \quad \text{[Head-Reduce]}$$

Reglas de traducción en paso pequeño

Tail de Lista

Error en lista vacía:

$$\frac{}{\langle \texttt{TailV}(\texttt{NilV}), \varepsilon\rangle \to \text{error}} \quad \text{[Tail-Nil-Error]}$$

Extraer cola NilV:

$$\frac{\text{isValue}(v)}{\langle \texttt{TailV}(\texttt{ConsV}(v, \texttt{NilV})), \varepsilon\rangle \to \langle \texttt{NilV}, \varepsilon\rangle} \quad \text{[Tail-Single]}$$

Extraer cola ConsV:

$$\frac{\text{isValue}(v)}{\langle \texttt{TailV}(\texttt{ConsV}(v, \texttt{ConsV}(h, t))), \varepsilon\rangle \to \langle \texttt{ConsV}(h, t), \varepsilon\rangle} \quad \text{[Tail-List]}$$

Reducir lista cuando cabeza no es valor:

$$\frac{\langle \texttt{ConsV}(e, r), \varepsilon\rangle \to \langle \texttt{ConsV}(e', r'), \varepsilon'\rangle \quad \neg\text{isValue}(e)}{\langle \texttt{TailV}(\texttt{ConsV}(e, r)), \varepsilon\rangle \to \langle \texttt{TailV}(\texttt{ConsV}(e', r')), \varepsilon\rangle} \quad \text{[Tail-Reduce-List]}$$

Reducir expresión general:

$$\frac{\langle e, \varepsilon\rangle \to \langle e', \varepsilon'\rangle}{\langle \texttt{TailV}(e), \varepsilon\rangle \to \langle \texttt{TailV}(e'), \varepsilon\rangle} \quad \text{[Tail-Reduce]}$$

4.9 Funciones y Aplicación

Reglas de traducción en paso pequeño

Creación de Clausuras

Lambda se convierte en clausura:

$$\frac{}{\langle \texttt{FunV}(p, c), \varepsilon\rangle \to \langle \texttt{ClosureV}(p, c, \varepsilon), \varepsilon\rangle} \quad \text{[Lambda-Closure]}$$

Reglas de traducción en paso pequeño

Aplicación de Funciones

Reducir función:

$$\frac{\langle f, \varepsilon\rangle \to \langle f', \varepsilon'\rangle \quad \neg\text{isValue}(f)}{\langle \texttt{AppV}(f, e), \varepsilon\rangle \to \langle \texttt{AppV}(f', e), \varepsilon'\rangle} \quad \text{[App-Fun]}$$

Reducir argumento:

$$\frac{\langle e, \varepsilon\rangle \to \langle e', \varepsilon'\rangle \quad \neg\text{isValue}(e)}{\langle \texttt{AppV}(\texttt{ClosureV}(p, c, \varepsilon_c), e), \varepsilon\rangle \to \langle \texttt{AppV}(\texttt{ClosureV}(p, c, \varepsilon_c), e'), \varepsilon'\rangle} \quad \text{[App-Arg]}$$

Beta-reducción:

$$\frac{\text{isValue}(v) \quad \varepsilon_{local} = (p, v) :: \varepsilon_c \quad \text{evalBody}(c, \varepsilon_{local}) = v_{result}}{\langle \texttt{AppV}(\texttt{ClosureV}(p, c, \varepsilon_c), v), \varepsilon\rangle \to \langle v_{result}, \varepsilon\rangle} \quad \text{[App-Beta]}$$

Error de tipo:

$$\frac{\text{isValue}(f) \quad f \neq \texttt{ClosureV}(\ldots)}{\langle \texttt{AppV}(f, e), \varepsilon\rangle \to \text{error}} \quad \text{[App-Error]}$$

4.10 Funciones Auxiliares

Reglas de traducción en paso pequeño

Predicado isValue

La función auxiliar isValue(e) determina si una expresión es un valor:

$$\begin{aligned}
\text{isValue}(\texttt{NumV}(n)) &= \text{True} \\
\text{isValue}(\texttt{BooleanV}(b)) &= \text{True} \\
\text{isValue}(\texttt{NilV}) &= \text{True} \\
\text{isValue}(\texttt{ClosureV}(p, c, \varepsilon)) &= \text{True} \\
\text{isValue}(\texttt{PairV}(v_1, v_2)) &= \text{isValue}(v_1) \wedge \text{isValue}(v_2) \\
\text{isValue}(\texttt{ConsV}(v_1, v_2)) &= \text{isValue}(v_1) \wedge \text{isValue}(v_2) \\
\text{isValue}(e) &= \text{False} \quad \text{(en cualquier otro caso)}
\end{aligned}$$

Reglas de traducción en paso pequeño

Función evalBody

La función evalBody(e, ε) evalúa completamente una expresión en un ambiente dado hasta obtener un valor:

$$\text{evalBody}(e, \varepsilon) = \begin{cases} e & \text{si isValue}(e) \\ \text{evalBody}(e', \varepsilon') & \text{si } \langle e, \varepsilon \rangle \to \langle e', \varepsilon' \rangle \end{cases}$$

Esta función es fundamental para la semántica call-by-value de la aplicación de funciones.

4.11 Observaciones sobre la Semántica

4.11.1 Estrategia de Evaluación

MiniLisp implementa una estrategia de evaluación *eager* (ansiosa) con paso de parámetros por valor (*call-by-value*). Esto significa que: Los argumentos de las funciones se evalúan completamente antes de realizar la aplicación (beta-reducción).

4.11.2 Manejo de Ambientes

Los ambientes (ε) son listas de pares $(variable, valor)$ que implementan alcance léxico (*lexical scoping*). La resolución de variables se realiza mediante la función lookup, que busca secuencialmente desde el binding más reciente.

Las clausuras capturan el ambiente en el momento de su creación, implementándo así el concepto de *closure* en el sentido funcional.

4.11.3 Propagación de Ambientes

En la mayoría de las reglas, el ambiente no se modifica durante la reducción ($\varepsilon' = \varepsilon$). La excepción principal es la aplicación de funciones, donde se crea un ambiente local extendido para evaluar el cuerpo de la función.

4.11.4 Manejo de Errores

El sistema cuenta con varios mecanismos de detección de errores:

- División por cero
- Raíz cuadrada de números negativos
- Operaciones sobre listas vacías (`head` y `tail`)
- Variables no definidas
- Aplicación de no-funciones

Todos estos casos resultan en una transición a un estado de error, deteniendo la evaluación e imprimiendo un mensaje de error si ocurre.

References

[1] M. Felleisen, R. B. Findler, and M. Flatt, *Syntactic Abstraction in Component Interfaces*, Springer, 1996.

[2] R. Harper, *Practical Foundations for Programming Languages*, Cambridge University Press, 2012.

[3] G. D. Plotkin, *A Structural Approach to Operational Semantics*, Technical Report DAIMI FN-19, Aarhus University, 1981.

5 Bibliografía