[Universidad del Valle]

Informe Proyecto Final
Interpretador SubOZ

Versión 1.0

Presentado por:
María Alejandra Pabón 1310263
Mayerly Suarez 1310284
Edwin Gamboa 1310233

Fundamentos de lenguajes de programación
Carlos A. Ramirez

Contenido

Pág.

Introducción

El presente informe incluye la información que se considera importante para comprender el proceso de construcción de un Interpretador para un subconjunto del lenguaje de programación Oz, al que hemos llamado SubOz. Se describe las estructuras de datos que se usaron, las decisiones de diseño que se tomaron para implementar cada componente, los métodos más relevantes y la metodología de trabajo utilizada para hacerlo.

El interpretador fue construido con base en el enunciado del proyecto definido por el profesor del curso y su asesoría.

1. Planeación

Antes de iniciar con la implementación propia del proyecto se construyó un cronograma actividades a realizar, en este se definieron fechas y responsables por cada actividad, dependiendo de la complejidad asignada a cada una de ellas.

Esta planeación se mantuvo hasta el final del proyecto, actualizando fechas, estados y responsables de acuerdo al desarrollo del mismo.

Las actividades se describirán en el resto del documento.

2. Definición de gramática

La gramática del interpretador fue construida principalmente gracias al enunciado del proyecto, construido por el profesor. En esta etapa se definieron los componentes que se podrían escribir con el lenguaje SubOz, estos son Programa, Cuerpo, Expresión, Primitiva, entre otros. A lo largo del informe se definirán claramente estos componentes.

Como elementos adicionales a los sugeridos en el enunciado del proyecto se incluyeron:

- Dos construcciones gramaticales que permitiera implementar expresiones condicionales (if-exp) que pudieran o no tener sentencia else:

```
(fin ("else" cuerpo "end")
         else)
(fin ("end")
         end)
```

- Unas construcciones gramaticales para los nombres de los campos de un registro de tal forma que estos pudieran ser átomos o enteros.

```
(campo (atomo)
           campo-1)
(campo (entero)
           campo-2)
```

3. Tipos y estructuras de datos

Las estructuras de datos que se utilizaron para implementar los diferentes componentes del interpretador fueron principalmente las listas y vectores de Scheme.

3.1 Ambiente

El ambiente es una estructura en la que se almacenan las variables de los programas que se crean con el lenguaje y la posición en el store que se les ha asignado. La definición del tipo de dato en Scheme para este es:

```
(define-datatype environment environment?
  (empty-env-record)
  (extended-env-record (syms (list-of symbol?))
                       (posiciones vector?)
                       (env environment?)))
```

La estructura para almacenar las variables es una lista, dado que estas no cambian después de ser creadas. Inicialmente para almacenar las posiciones se escogió un vector, dado que en ese momento se pensaba que las cuando una variable fuera asignada a un valor que ya estuviera en el store, se actualizaría su posición en el ambiente por la del valor. Sin embargo tras incluir la estructura variable, cambios como este se gestionan directamente en el store. En este sentido ya no se requiere un vector para las posiciones, pero no se cambió por listas puesto que ya se había avanzado en un alto porcentaje de la implementación del proyecto, el tiempo disponible para completarlo ya era reducido y es de entender que un cambio sobre el ambiente requiere modificar muchas cosas en todo el código. Los métodos asociados al ambiente son:

- empty-env: función que crea un ambiente vacío.
- extend-env: función que crea un ambiente extendido, el cual tiene nuevas variables y nuevas posiciones, alcanzables desde el lugar donde se extiende.
- apply-env: función que busca un símbolo en un ambiente, devuelve la posición en el store asignada a ese símbolo.

3.2 Almacén (Store)

El ambiente es una estructura compartida por todos los elementos del lenguaje SubOz, y cuyo contenido debe estar disponible en todo momento y de forma completa y correcta. En este se almacenan los valores de cada una de las variables de los programas que se escriben, estos valores pueden ser modificados una solo vez siempre que su valor actual sea ” _”. su tamaño es variable ya que se crea una nueva posición por cada elemento nuevo que lo requiera. Para lograr el funcionamiento esperado se utilizó un vector de Scheme y dado que su tamaña debe cambiar, cada que se le agregan elementos. La definición del tipo de dato almacén es la siguiente:

```
(define-datatype store store?
  (a-store
   (vec-vals vector?)))
```

Los métodos principales del mismo son:

- extend-store: función que agrega un nuevo elemento al store, para esto crea un store extendido y lo retorna. El store extendido es un vector con el contenido anterior y una posición adicional que contiene el nuevo elemento. Para lograr un store de tamaño variable se utilizó el método *vector-append* de Scheme.
- apply-store: devuelve el valor almacenado en una posición del store.
- set-store: modifica el valor almacenado en una posición del store.
- store-size: permite conocer el tamaño de un store. Equivale al número de elementos almacenados en el store.
- simbolo-no-asignado?: permite saber si un símbolo es igual al símbolo *(“_”)* que indica que una variable no ha sido asignada.
- extend-store-varios: cumple la misma función que extend-store para para varios elementos. Devuelve una lista cuyo car es el store-extendido y cdr es el listado de las posiciones que fueron asignadas a esos elementos.

Un hecho a destacar es que inicialmente el store no era manejado de forma global y no generaba inconvenientes, sin embargo mientras se implementaba el algoritmo de unificación fue posible identificar problemas puesto que al hacer cambios sobre el mismo, estos no eran percibidos al momento de evaluar otras expresiones de un cuerpo.

3.3 Registros

Para la implementación de los registros se utilizaron los tipos de dato *campo* y *a-var,* el primero para guardar los nombres de los campos de un registro y el segundo para mantener la posición del valor de cada campo en el store, la definición del tipo de dato registro es:

```
(define-datatype registro registro?
  (empty-record
   (label symbol?))
  (record-exp
   (label symbol?)
   (campos (list-of campo?))
   (posiciones vector?)))
```

Un ejemplo de un registro es:

```
miregistro(campo1:X campo2:4) -->

      #(struct:record-exp
            |miregistro(|
            (#(struct:campo-1    campo1)    #(struct:campo-1    campo2))
            #(#(struct:a-var 2) #(struct:a-var 5)))
```

3.3.1 Crear registro

Cuando se evalúa una expresión equivalente a un registro se crea un registro de la forma del tipo de dato registro de la siguiente manera. Primero en el store se guardan los valores de los campos que no estaban almacenados, es decir, los valores diferentes a una variable (X, Y, etc.). Luego se toman las posiciones asignadas a estos valores nuevos que se almacenaron y se reúnen con las posiciones que ya existían, las cuales corresponden a las posiciones del store de los valores de las variables (X, Y, etc.), para así formar (a-var pos) de cada una y asignarle a cada campo su (a-var pos) respectivo. Luego se arma el registro con la etiqueta, la lista de campos y el vector de (a-var-pos) que apunta a cada valor.

Los métodos que se utilización para implementar la creación de un registro fueron:

- crear-registro: función principal que se encarga de armar el registro y crearlo.
- reunir-valores-no-almacenados-en-store: función que reúne todos los valores de los campos que no están almacenados en ese momento, es decir, los valores diferentes a una variables (X, Y, etc.) los toma de la lista de valores que llega como parámetro a crear-registro.
- llenar-store-con-valores-campo-no-almacenados: función que almacena en el store la lista que retorna reunir-valores-no-almacenados-en-store.
- reunir-todas-posiciones-campos: función que reúne todas las posiciones en el store de los valores de cada campo y al utilizar crear-registro.
- reunir-a-var-de-campos: función que reúne los (a-var pos) de cada campo para armar el vector de varios (a-var pos) .
- asignar-a-var-a-campos: función que asigna a cada campo un (a-var pos) que apunta a su valor, utiliza a la función reunir-a-var-de-campos

3.3.2 Listas

Forma 1

Dado que las listas se deben guardar como registros, lo primero que se hace es construir una expresión de tipo record-exp1 con una etiqueta 'l, un primer campo 'h, que guarda el primer valor de la lista y un segundo campo 't, que guarda una expresión record-exp1 con dos campos 'h y 't, donde 'h guarda el segundo elemento y y otro registro, así hasta llegar a un record-exp donde 'h sea el último elemento de la lista y 't el átomo nil.
Dado que record-exp1 es una expresión al final se llama al método eval-.expression para que lo trate como un registro normal y lo almacene en el store correctamente.

Forma 2

Opcional, se explicara en sustentación.

3.3.3 Obtener valor campo

El método obtener-valor-campo permite buscar el valor de un campo de un registro en el store. Dado que en los registros se guardan estructuras de tipo *a-var* entonces se busca en el store el valor que en la posición del a-var de cada campo. Si el registro es vacío o el campo no se encuentra en el registro arroja un error.

3.3.4 Registros iguales para unificación

El método registros-iguales-para-unificar? permite comparar dos valores teniendo en cuenta lo siguiente: los dos valores deben ser un registro, con igual etiqueta, número de campos y nombres de campos. De esta manera se pueden unificar los valores de los campos de cada registro que tengan igual nombre de campo.

3.4 Procedimientos

En la declaración de procedimientos fue necesario construir un tipo de dato Procval, que contiene dos variantes:

- closure (corresponde a expresiones de tipo **proc**)
- función (para el tipo de expresiones **fun**).

Las variantes tienen exactamente las mismas variables: nombre, vars, cuerpo y env; sin embargo, fue factible crearlo para facilitar su almacenamiento en el store. Adicionalmente, se requería para diferenciar que la aplicaciones de funciones pueden retornar cualquier tipo de expresión, mientras que la aplicación de la variante closure no retorna ningún valor operable (en el caso particular de la implementación, el tipo de dato void, del cual se hablará en detalle más adelante).

```
(define-datatype procval procval?
  (closure
   (nom nombre-proc?)
   (vars (list-of symbol?))
   (cuerpo cuerpo?)
   (env environment?))
  (funcion
   (nom nombre-proc?)
   (vars (list-of symbol?))
   (cuerpo cuerpo?)
   (env environment?)))
```

3.5 Variable

La estructura *a-var* se utiliza para mantener las posiciones de los valores de variables o campos de un registro, la necesidad de su creación resultó durante el proceso de implementación del algoritmo de unificación y específicamente para gestionar casos de unificación como *set A=B* y luego *set B=A.* La definición del tipo de dato variable es:

```
(define-datatype variable variable?
      (a-var (posicion scheme-value?)))
```

3.6 Vacío

Se decidió definir un tipo de dato vacío para devolver en funciones que no requieran devolver un valor operable, como es el caso de la aplicación de proc, el tipo de expresión skip y el for. Fue una decisión de diseño escoger un valor que no pudiera ser usado por otro tipo de expresión, para evitar ambigüedades en la construcción del algoritmo de unificación o en cualquier otro caso. La definición del tipo de dato es la siguiente:

```
(define-datatype vacio vacio?

(void))
```

4. Cuerpo

Un cuerpo está formado por varias expresiones y al ser evaluado debe devolver el valor de la evaluación de la última expresión. Para esto fue creado el método ***eval-cuerpo,*** que evalúa cada una de las expresiones en el ambiente actual y retorna el valor de la última evaluación.

Este método es útil en aplicación de procedimientos, en evaluación de expresiones condicionales, ciclos, entre otros.

5. Primitivas y aplicación de primitivas

Para lograr que todas las primitivas funcionaran correctamente fue necesario implementar los siguientes métodos auxiliares:

Función	**Primitivas relacionadas**
apply-multi-prim: función auxiliar que evalúa el caso en el que una primitiva se deba aplicar a dos o más argumentos.	sum-prim, mult-prim.
apply-multi-prim-aux: calcula o aplica la operación de la primitiva sobre los argumentos después de haber evaluado su tamaño correcto en apply-multi-prim.	sum-prim, mult-prim.
apply-multi-prim-aux-tipo:	sum-prim, mult-prim.
apply-doble-prim: función auxiliar que aplica una primitiva a dos argumentos.	sub-prim, div-prim.
apply-doble-prim-unificacion: función auxiliar que aplica una unificación a dos términos.	unif-prim
apply-isdet-prim: función auxiliar que verifica que la primitiva isdet? solo reciba un argumento.	isdet-prim
apply-isdet-prim-aux: función auxiliar que aplica la primitiva isdet? sobre una variable	isdet-prim
isdet? retorna true si una variable está asignada a algún valor y false si no.	isdet-prim
apply-isfree-prim: función auxiliar que verifica que la primitiva isfree? solo reciba un argumento.	isfree-prim
apply-isfree-prim-aux: función auxiliar que aplica la primitiva isfree? sobre una variable.	isfree-prim
apply-multi-type-prim: función auxiliar que aplica una primitiva a dos argumentos, que pueden ser números, símbolos o strings de scheme.	menor-prim, meneq-prim, mayor-prim, mayig-prim, igual-prim.

atomo-booleano: función auxiliar que convierte un átomo (true o false) ingresado como parámetro en su correspondiente representación de booleanos para Scheme, facilitando el uso de and y or del mismo lenguaje.	orelse, andthen

6. Aplicación de funciones y procedimientos

Para la aplicación de procedimientos y funciones se creó el método principal *apply-procedure,* que recibe como parámetros un tipo de dato *procval* y una lista de argumentos, hace uso de tipo de dato *procval* y *void*, además del auxiliar del almacén *extend-store-varios* y el método siguientes métodos auxiliares *eval-cuerpo*:

```
(apply-procedure procedimiento args)
```

Finalmente, la aplicación de procedimientos devuelve en el caso de funciones *(fun)* la expresión que devuelva el método eval-cuerpo y para closure *(proc)* el tipo de dato vacío (*void)*.

7. Variables locales

Las variables creadas con la sentencia *local* se guardan en un ambiente extendido y son accesibles solamente desde su cuerpo. Inicialmente son variables sin ligar y apuntan a una posición cuyo valor es *"_".* La variable "_" es creada en el ambiente inicial y su valor en el store es "_"

8. Condicional

Para lograr un buen funcionamiento de los condicionales, se implementaron los átomos '*false* y *'true* y se construyeron los métodos *true-value?* y *false-value?* que determinan si un valor es o no el átomo *'true* o el átomo *'false*. Con esto se logró que la condición del *if* fuera de tipo booleano y se tratará como tal. En caso de que la condición no se de tipo booleano se devuelve un error.

Para evaluar una expresión condicional primero se evalúa la condición del if, si está vale *'true* entonces se evalúa el primer cuerpo del if, de lo contrario se verifica si hay o no un cuerpo para el *else,* si lo hay se evalúa y si no se devuelve *void.*

9. Ciclos

Para ejecutar los ciclos *for* se valida que los límites del mismo sean correctos, es decir que sean números y que el valor inicial sea menor que el final. Luego se procede a hacer un llamado a la función auxiliar *do-for,* que extiende el ambiente actual con la variable asociada al *for* y cuyo valor en el almacén es el que corresponde a la iteración actual. Luego se hace un llamado al método *eval-cuerpo* para evaluar el cuerpo del for. Se hace lo mismo hasta que terminen las iteraciones.

10. Chequeo dinámico de tipos

Para la aplicación de primitivas se realiza un chequeo dinámico de tipos con ayuda de predicados de scheme, para garantizar que se ingresen la cantidad de argumentos correctos y con el tipo correcto, en la siguiente tabla se relaciona los chequeos de tipo realizados:

Chequeo	**Primitivas asociadas**	**Observaciones**
integer?	+, -, *, /	Para garantizar que se hagan primitivas aritméticas sobre solo enteros.
inexact?	+, -, *, /	Para garantizar que se hagan primitivas aritméticas sobre solo flotantes.
boolean?	orelse, andthen	Para garantizar que se hagan primitivas lógicas sobre booleanos.
number?	menor, meneq, mayor, mayig, igual	Para garantizar que se hagan primitivas de comparación sobre solo números.
symbol?	menor, meneq, mayor, mayig, igual	Para garantizar que se hagan primitivas de comparación sobre solo átomos.

11. Unificación

El algoritmo de unificación se encarga de unificar los valores y las variables en dos términos. Se tomaron como términos los siguientes:

Término	Descripción de la expresión
var-exp	Variable anónima: '_ o palabra que empieza en letra mayúscula. Esta variable se almacena en el ambiente. Evaluarla devuelve su valor en el store
entero-exp	Número entero. Evaluarlo devuelve un número entero.
flotante-exp	Número flotante. Evaluarlo devuelve un número flotante.
record-exp1	Registro no vació. Evaluarlo devuelve el registro creado.
record-exp2	Registro vació que corresponde a un átomo. Evaluarlo devuelve el registro vacío creado.
asg-record-exp	Acceso al valor de un campo de un registro. Evaluarlo devuelve ese valor.
list-exp	Lista. Evaluarla devuelve un registro especial que representa una lista.
proc-exp	Procedimiento. Evaluarlo devuelve un closure.
fun-exp	Función. Evaluarlo devuelve una función (closure de función).
app-exp	Aplicación de procedimiento o de función. Si es una aplicación de procedimiento, evaluarla devuelve (void). Si es una aplicación de función evaluarla devuelve el valor que retorne la función.
if-exp	Condicional. Evaluarlo devuelve un valor según lo que suceda en el condicional.
for-exp	Ciclo. Evaluarlo devuelve (void)
prim-exp	Aplicación de primitiva. Evaluarla devuelve el resultado de la aplicación de la primitiva.

Entonces se tomó que un término igualado a otro dependiendo del caso se pueden unificar o no según los siguientes casos:

- Las variables solo se pueden asignar una vez
- Si t1 y t2 son el mismo término, es decir son el mismo valor o devuelven el mismo valor, entonces se retorna void sino error.
- Si t1 es una variable no asignada y t2 es un valor o devuelve un valor entonces se unifican t1 y t2. Y viceversa.
- Si t1 es una variable asignada y t2 es un valor o devuelve un valor entonces se genera un error. Y viceversa.
- Si t1 está igualado a un valor al cual no se debe igualar entonces se genera un error. Y viceversa.
- Si t1 es una variable no asignada y t2 es una variable asignada se modifica pos en el store de t1 por pos de t2 donde esa pos esta guardada en un (a-var pos)
- Si t1 es una variable asignada y t2 es una variable no asignada se modifica pos en el store de t2 por pos de t1 donde esa pos esta guardada en un (a-var pos)
- Si t1 es una variable no asignada y t2 es una variable no asignada t1 y t2 se convierten en la misma variable, es decir apuntan a la misma posición en el store
- Si t1 es una variable ya asignada y t2 es una variable ya asignada y sus valores son registros e iguales se realiza unificación de registros, de lo contrario se genera un error
- Si se iguala un término a algo que no sea un término se genera un error.

Aclaraciones:

- Si los dos términos son iguales la unificación no tiene problema, esto lo tomamos como que se devolviera un (void), puesto que para nosotros dos términos iguales son dos términos con el mismo valor. Entonces si se llaman a un término o al otro van a devolver el mismo valor.
- Si los dos términos son diferentes la unificación falla. Esto es por ejemplo set 5=6
- Para el caso de la unificación de dos variables no asignadas, se tomó la decisión de que si tengo set X=Y, siendo X y X variables no asignadas,

entonces se convierten en la misma variable. Primero se extiende el store con la variable var-no-asig (`_) y luego se hace que X, Y apunten a ese valor con su respectiva posición. De esta manera se convierten en la misma variable. Pero se genera un problema al hacer set X=Y Y=Z, siendo X, Y y Z variables no asignadas, pues la unificación de los tres se pierde. Por esto decidimos dejar que solo funcionara esta parte de unificación para dos variables y no más de dos.

- En la unificación de registros se tomó la siguiente decisión: solamente se unifican valores de campos con el mismo nombre, siempre y cuando estos nombres de campos estén en el mismo orden tanto en un registro como en el otro. También debido a que al crear un registro se tiene es una lista de campos y un vector de varios (a-var pos) que apunta a la posición en el store del valor, entonces decidimos crear un método de unificación especial, donde se reciban valores ya evaluados , diferente al unificación principal que recibe expresiones en sintaxis abstracta. Los métodos relacionados a esta unificación de valores de campos de registro son:

    - ❏ unificar-registros: función que unifica los valores de los campos de los registros, con ayuda de las dos funciones auxiliar-unificar-reg y unificación-valores-de-a-var
    - ❏ auxiliar-unificar-reg: función que recorre los campos de cada registro en orden y va unificando sus valores con la función unificacion-valores-de-a-var
    - ❏ unificacion-valores-de-a-var: función que unifica los valores de los campos de un registro

- Los métodos auxiliares al método principal unificación fueron

    - ❏ auxiliarUnificacion: función que unifica dos términos, donde uno es una variable y el otro es cualquier otro término que no sea una variable, sin importar el orden de los términos.
    - ❏ auxiliarUnificacionProcFun: funcion que unifica dos términos, donde uno es una variable y el otro es un procedimiento o función, sin importar el orden de los términos.
    - ❏ auxiliar-unificacion-app: funcion que unifica dos términos, donde uno es una variable y el otro es una aplicación de procedimiento o función, sin importar el orden de los términos.
    - ❏ iguales-terminos?: función que determina si dos términos son iguales.