Laboratorio de Lenguajes de Programación

Funciones de primera clase

Manuel Soto Romero

Universidad Nacional Autónoma de México
Facultad de Ciencias

28 de septiembre de 2016

Objetivos

Objetivos

- Recordar la idea de endulzar el lenguaje, es decir, azúcar sintáctica.

Objetivos

- Recordar la idea de endulzar el lenguaje, es decir, azúcar sintáctica.
- Presentar la clasificación de funciones en cuanto a los lenguajes de programación se refiere.

Objetivos

- Recordar la idea de endulzar el lenguaje, es decir, azúcar sintáctica.
- Presentar la clasificación de funciones en cuanto a los lenguajes de programación se refiere.
- Extener el lenguaje de `FWAE` para que use funciones de primera clase.

Objetivos

- Recordar la idea de endulzar el lenguaje, es decir, azúcar sintáctica.
- Presentar la clasificación de funciones en cuanto a los lenguajes de programación se refiere.
- Extener el lenguaje de `FWAE` para que use funciones de primera clase.
- Endulzar el lenguaje `FWAE` para una mejor manipulación.

Objetivos

- Recordar la idea de endulzar el lenguaje, es decir, azúcar sintáctica.
- Presentar la clasificación de funciones en cuanto a los lenguajes de programación se refiere.
- Extener el lenguaje de `FWAE` para que use funciones de primera clase.
- Endulzar el lenguaje `FWAE` para una mejor manipulación.
- Repasar la técnica de sustitución diferida mediante ambientes.

Azúcar sintáctica

Azúcar sintáctica

¿Qué hace la siguiente expresión en Java?

```
if (!iteradorValido())
   return null;
return iterador.elemento;
```

Azúcar sintáctica

¿Qué hace la siguiente expresión en Java?

```
if (!iteradorValido())
   return null;
return iterador.elemento;
```

¿Qué hace la siguiente expresión en Java?

```
return !iteradorValido() ? null : iterador.elemento;
```

Azúcar sintáctica

¿Qué hace la siguiente expresión en Java?

```
if (!iteradorValido())
   return null;
return iterador.elemento;
```

¿Qué hace la siguiente expresión en Java?

```
return !iteradorValido() ? null : iterador.elemento;
```

La segunda expresión es una «simplificación» de la primera. Es más amistosa para el usuario, pues escribe todo de una forma más simple.

Azúcar sintáctica

¿Qué hace la siguiente expresión en Java?

```
if (!iteradorValido())
   return null;
return iterador.elemento;
```

¿Qué hace la siguiente expresión en Java?

```
return !iteradorValido() ? null : iterador.elemento;
```

La segunda expresión es una «simplificación» de la primera. Es más amistosa para el usuario, pues escribe todo de una forma más simple.

En lenguajes de programación, usamos este tipo de simplificaciones para «endulzar el lenguaje», de esta forma decimos que una expresión es azúcar sintáctica de otra.

Endulzando FWAE

Endulzando FWAE

Debido a que tenemos funciones y aplicación de funciones como dos primitivas distintas, podemos combinarlas para dar el funcionamiento de with como un caso especial. Cada que encontremos una expresión de la forma:

```
{with {var named} body}
```

Endulzando FWAE

Debido a que tenemos funciones y aplicación de funciones como dos primitivas distintas, podemos combinarlas para dar el funcionamiento de with como un caso especial. Cada que encontremos una expresión de la forma:

```
{with {var named} body}
```

La reemplazaremos por otra de la forma:

```
{{fun {var} body} named}
```

Endulzando FWAE

Debido a que tenemos funciones y aplicación de funciones como dos primitivas distintas, podemos combinarlas para dar el funcionamiento de with como un caso especial. Cada que encontremos una expresión de la forma:

```
{with {var named} body}
```

La reemplazaremos por otra de la forma:

```
{{fun {var} body} named}
```

Este «endulzamiento» es más claro para el usuario. Para nosotros como diseñadores del lenguaje, es más fácil operar las primitivas como una sola y no por separado. Podemos desaparecer el `with` de nuestra gramática:

```
(define-type FAE
   [num (n number?)]
   [id (name symbol?)]
   [fun (param symbol?) (body FAE?)]
   [app (fun FAE?) (arg-expr FAE?)]
   [binop (f procedure?) (l FAE?) (r FAE?)])
```

Endulzando FWAE

Debido a que tenemos funciones y aplicación de funciones como dos primitivas distintas, podemos combinarlas para dar el funcionamiento de with como un caso especial. Cada que encontremos una expresión de la forma:

```
{with {var named} body}
```

La reemplazaremos por otra de la forma:

```
{{fun {var} body} named}
```

Este «endulzamiento» es más claro para el usuario. Para nosotros como diseñadores del lenguaje, es más fácil operar las primitivas como una sola y no por separado. Podemos desaparecer el `with` de nuestra gramática:

```
(define-type FAE
   [num (n number?)]
   [id (name symbol?)]
   [fun (param symbol?) (body FAE?)]
   [app (fun FAE?) (arg-expr FAE?)]
   [binop (f procedure?) (l FAE?) (r FAE?)])
```

Convertiremos las expresiones del usuario a `FWAE` y luego las endulzaremos pasándolas a `FAE`.

```
(define-type FWAE
   [numS (n number?)]
   [idS (name symbol?)]
   [withS (name symbol?) (named-expr FWAE?) (body FWAE?)]
   [funS (param symbol?) (body FWAE?)]
   [appS (fun-expr FWAE?) (arg-expr FWAE?)]
   [binopS (f procedure?) (l FWAE?) (r FWAE?)])

(define-type FAE
   [num (n number?)]
   [id (name symbol?)]
   [fun (param symbol?) (body FAE?)]
   [app (fun-expr FAE?) (arg-expr FAE?)]
   [binop (f procedure?) (l FAE?) (r FAE?)])
```

```
(define-type FWAE
   [numS (n number?)]
   [idS (name symbol?)]
   [withS (name symbol?) (named-expr FWAE?) (body FWAE?)]
   [funS (param symbol?) (body FWAE?)]
   [appS (fun-expr FWAE?) (arg-expr FWAE?)]
   [binopS (f procedure?) (l FWAE?) (r FWAE?)])

(define-type FAE
   [num (n number?)]
   [id (name symbol?)]
   [fun (param symbol?) (body FAE?)]
   [app (fun-expr FAE?) (arg-expr FAE?)]
   [binop (f procedure?) (l FAE?) (r FAE?)])

(define (desugar expr)
   (match expr
```

```
(define-type FWAE
   [numS (n number?)]
   [idS (name symbol?)]
   [withS (name symbol?) (named-expr FWAE?) (body FWAE?)]
   [funS (param symbol?) (body FWAE?)]
   [appS (fun-expr FWAE?) (arg-expr FWAE?)]
   [binopS (f procedure?) (l FWAE?) (r FWAE?)])

(define-type FAE
   [num (n number?)]
   [id (name symbol?)]
   [fun (param symbol?) (body FAE?)]
   [app (fun-expr FAE?) (arg-expr FAE?)]
   [binop (f procedure?) (l FAE?) (r FAE?)])

(define (desugar expr)
   (match expr
      [(numS n) (num n)]
```

```
(define-type FWAE
   [numS (n number?)]
   [idS (name symbol?)]
   [withS (name symbol?) (named-expr FWAE?) (body FWAE?)]
   [funS (param symbol?) (body FWAE?)]
   [appS (fun-expr FWAE?) (arg-expr FWAE?)]
   [binopS (f procedure?) (l FWAE?) (r FWAE?)])

(define-type FAE
   [num (n number?)]
   [id (name symbol?)]
   [fun (param symbol?) (body FAE?)]
   [app (fun-expr FAE?) (arg-expr FAE?)]
   [binop (f procedure?) (l FAE?) (r FAE?)])

(define (desugar expr)
   (match expr
      [(numS n) (num n)]
      [(idS name) (id name)]
```

FWAE a FAE

```
(define-type FWAE
   [numS (n number?)]
   [idS (name symbol?)]
   [withS (name symbol?) (named-expr FWAE?) (body FWAE?)]
   [funS (param symbol?) (body FWAE?)]
   [appS (fun-expr FWAE?) (arg-expr FWAE?)]
   [binopS (f procedure?) (l FWAE?) (r FWAE?)])

(define-type FAE
   [num (n number?)]
   [id (name symbol?)]
   [fun (param symbol?) (body FAE?)]
   [app (fun-expr FAE?) (arg-expr FAE?)]
   [binop (f procedure?) (l FAE?) (r FAE?)])

(define (desugar expr)
   (match expr
      [(numS n) (num n)]
      [(idS name) (id name)]
      [(withS name named-expr body)
         (app (fun name (desugar body)) (desugar named-expr))]
```

```
(define-type FWAE
   [numS (n number?)]
   [idS (name symbol?)]
   [withS (name symbol?) (named-expr FWAE?) (body FWAE?)]
   [funS (param symbol?) (body FWAE?)]
   [appS (fun-expr FWAE?) (arg-expr FWAE?)]
   [binopS (f procedure?) (l FWAE?) (r FWAE?)])

(define-type FAE
   [num (n number?)]
   [id (name symbol?)]
   [fun (param symbol?) (body FAE?)]
   [app (fun-expr FAE?) (arg-expr FAE?)]
   [binop (f procedure?) (l FAE?) (r FAE?)])

(define (desugar expr)
   (match expr
      [(numS n) (num n)]
      [(idS name) (id name)]
      [(withS name named-expr body)
         (app (fun name (desugar body)) (desugar named-expr))]
      [(funS param body) (fun param (desugar body))]
```

FWAE a FAE

```
(define-type FWAE
   [numS (n number?)]
   [idS (name symbol?)]
   [withS (name symbol?) (named-expr FWAE?) (body FWAE?)]
   [funS (param symbol?) (body FWAE?)]
   [appS (fun-expr FWAE?) (arg-expr FWAE?)]
   [binopS (f procedure?) (l FWAE?) (r FWAE?)])

(define-type FAE
   [num (n number?)]
   [id (name symbol?)]
   [fun (param symbol?) (body FAE?)]
   [app (fun-expr FAE?) (arg-expr FAE?)]
   [binop (f procedure?) (l FAE?) (r FAE?)])

(define (desugar expr)
   (match expr
      [(numS n) (num n)]
      [(idS name) (id name)]
      [(withS name named-expr body)
         (app (fun name (desugar body)) (desugar named-expr))]
      [(funS param body) (fun param (desugar body))]
      [(appS fun-expr arg-expr) (app (desugar fun-expr) (arg-expr))]
```

FWAE a FAE

```
(define-type FWAE
   [numS (n number?)]
   [idS (name symbol?)]
   [withS (name symbol?) (named-expr FWAE?) (body FWAE?)]
   [funS (param symbol?) (body FWAE?)]
   [appS (fun-expr FWAE?) (arg-expr FWAE?)]
   [binopS (f procedure?) (l FWAE?) (r FWAE?)])

(define-type FAE
   [num (n number?)]
   [id (name symbol?)]
   [fun (param symbol?) (body FAE?)]
   [app (fun-expr FAE?) (arg-expr FAE?)]
   [binop (f procedure?) (l FAE?) (r FAE?)])

(define (desugar expr)
   (match expr
      [(numS n) (num n)]
      [(idS name) (id name)]
      [(withS name named-expr body)
         (app (fun name (desugar body)) (desugar named-expr))]
      [(funS param body) (fun param (desugar body))]
      [(appS fun-expr arg-expr) (app (desugar fun-expr) (arg-expr))]
      [(binopS f l r) (binop f (desugar l) (desugar r))]))
```

Internamente...

Internamente...

1. Escribimos una expresión en sintaxis concreta:

```
{with {a 2} {+ a a}}
```

Internamente...

1. Escribimos una expresión en sintaxis concreta:

```
{with {a 2} {+ a a}}
```

2. Realizamos el árbol de sintaxis abstracta en `FWAE`:

```
(with 'a (num 2) (binop + (id 'a) (id 'a)))
```

Internamente...

1. Escribimos una expresión en sintaxis concreta:

```
{with {a 2} {+ a a}}
```

2. Realizamos el árbol de sintaxis abstracta en `FWAE`:

```
(with 'a (num 2) (binop + (id 'a) (id 'a)))
```

3. Le quitamos el endulzamiento, es decir, pasamos la expresión a `FAE`:

```
(app (fun 'a (binop + (id 'a) (id 'a))) (num 2))
```

Internamente...

1. Escribimos una expresión en sintaxis concreta:

```
{with {a 2} {+ a a}}
```

2. Realizamos el árbol de sintaxis abstracta en `FWAE`:

```
(with 'a (num 2) (binop + (id 'a) (id 'a)))
```

3. Le quitamos el endulzamiento, es decir, pasamos la expresión a `FAE`:

```
(app (fun 'a (binop + (id 'a) (id 'a))) (num 2))
```

Esto es más fácil de interpretar

¿Dónde estamos parados?

¿Dónde estamos parados?

El lenguaje FWAE estudiado en la sesiones de laboratorio pasadas tiene la siguiente sintaxis abstracta (agregamos la `S` para indicar que está endulzado):

¿Dónde estamos parados?

El lenguaje FWAE estudiado en la sesiones de laboratorio pasadas tiene la siguiente sintaxis abstracta (agregamos la `S` para indicar que está endulzado):

```
(define-type FWAE
   [numS (n number?)]
   [binopS (f procedure?) (l FWAE?) (r FWAE?)]
   [withS (name symbol?) (named-expr FWAE?) (body FWAE?)]
   [idS (name symbol?)]
   [funS (param symbol?) (body FWAE?)]
   [appS (fun-expr FWAE?) (arg-expr FWAE?)])
```

¿Dónde estamos parados?

El lenguaje FWAE estudiado en la sesiones de laboratorio pasadas tiene la siguiente sintaxis abstracta (agregamos la `S` para indicar que está endulzado):

```
(define-type FWAE
   [numS (n number?)]
   [binopS (f procedure?) (l FWAE?) (r FWAE?)]
   [withS (name symbol?) (named-expr FWAE?) (body FWAE?)]
   [idS (name symbol?)]
   [funS (param symbol?) (body FWAE?)]
   [appS (fun-expr FWAE?) (arg-expr FWAE?)])
```

El interprete construido para este lenguaje no hace realmente una evaluación de funciones:

¿Dónde estamos parados?

El lenguaje FWAE estudiado en la sesiones de laboratorio pasadas tiene la siguiente sintaxis abstracta (agregamos la `S` para indicar que está endulzado):

```
(define-type FWAE
   [numS (n number?)]
   [binopS (f procedure?) (l FWAE?) (r FWAE?)]
   [withS (name symbol?) (named-expr FWAE?) (body FWAE?)]
   [idS (name symbol?)]
   [funS (param symbol?) (body FWAE?)]
   [appS (fun-expr FWAE?) (arg-expr FWAE?)])
```

El interprete construido para este lenguaje no hace realmente una evaluación de funciones:

```
[(funS bound-id bound-body) "#<function>"]
```

¿Dónde estamos parados?

El lenguaje FWAE estudiado en la sesiones de laboratorio pasadas tiene la siguiente sintaxis abstracta (agregamos la S para indicar que está endulzado):

```
(define-type FWAE
   [numS (n number?)]
   [binopS (f procedure?) (l FWAE?) (r FWAE?)]
   [withS (name symbol?) (named-expr FWAE?) (body FWAE?)]
   [idS (name symbol?)]
   [funS (param symbol?) (body FWAE?)]
   [appS (fun-expr FWAE?) (arg-expr FWAE?)])
```

El interprete construido para este lenguaje no hace realmente una evaluación de funciones:

```
[(funS bound-id bound-body) "#<function>"]
```

¿Qué limitaciones da esta forma de *interpretar* funciones?

Taxonomía de funciones

Taxonomía de funciones

En el mundo de los lenguajes de programación existe una clasificación o taxonomía que describe cómo se comportan las funciones de un determinado lenguaje.

Taxonomía de funciones

En el mundo de los lenguajes de programación existe una clasificación o taxonomía que describe cómo se comportan las funciones de un determinado lenguaje.

- **Primer orden.** Las funciones no son valores en el lenguaje.

Taxonomía de funciones

En el mundo de los lenguajes de programación existe una clasificación o taxonomía que describe cómo se comportan las funciones de un determinado lenguaje.

- **Primer orden.** Las funciones no son valores en el lenguaje.
- **Orden superior.** Las funciones pueden regresar otras funciones como valores.

Taxonomía de funciones

En el mundo de los lenguajes de programación existe una clasificación o taxonomía que describe cómo se comportan las funciones de un determinado lenguaje.

- **Primer orden.** Las funciones no son valores en el lenguaje.
- **Orden superior.** Las funciones pueden regresar otras funciones como valores.
- **Primera clase.** Las funciones son tratadas como cualquier otro valor dentro del lenguaje. Pueden pasarse como parámetro a otras funciones, pueden regresarse y almacenarse.

¿Qué tipo de funciones usa FWAE?

¿Qué tipo de funciones usa FWAE?

¿Podemos escribir la siguiente llamada en el intérprete de FWAE? ¿Qué regresa?

```
(λ) {fun {x} {fun {y} {pow x y}}}
```

¿Qué tipo de funciones usa FWAE?

¿Podemos escribir la siguiente llamada en el intérprete de FWAE? ¿Qué regresa?

```
(λ) {fun {x} {fun {y} {pow x y}}}
```

¿Las funciones de FWAE son de orden superior?

¿Qué tipo de funciones usa FWAE?

¿Podemos escribir la siguiente llamada en el intérprete de FWAE? ¿Qué regresa?

```
(λ) {fun {x} {fun {y} {pow x y}}}
```

¿Las funciones de FWAE son de orden superior?

¿Podemos escribir la siguiente llamada en el intérprete de FWAE? ¿Qué regresa?

```
(λ) {{fun {foo} {foo 3}} {fun {x} {pow x 2}}}
```

¿Qué tipo de funciones usa FWAE?

¿Podemos escribir la siguiente llamada en el intérprete de FWAE? ¿Qué regresa?

```
(λ) {fun {x} {fun {y} {pow x y}}}
```

¿Las funciones de FWAE son de orden superior?

¿Podemos escribir la siguiente llamada en el intérprete de FWAE? ¿Qué regresa?

```
(λ) {{fun {foo} {foo 3}} {fun {x} {pow x 2}}}
```

¿Las funciones de FWAE son de primera clase?

¿Qué tipo de funciones usa FWAE?

¿Podemos escribir la siguiente llamada en el intérprete de FWAE? ¿Qué regresa?

```
(λ) {fun {x} {fun {y} {pow x y}}}
```

¿Las funciones de FWAE son de orden superior?

¿Podemos escribir la siguiente llamada en el intérprete de FWAE? ¿Qué regresa?

```
(λ) {{fun {foo} {foo 3}} {fun {x} {pow x 2}}}
```

¿Las funciones de FWAE son de primera clase?

Las funciones de FWAE son de primer orden en esta versión

Funciones de primera clase en FWAE

Funciones de primera clase en FWAE

¿Qué tipo de valores debe regresar nuestro intérprete si queremos funciones de primera clase?

Funciones de primera clase en FWAE

¿Qué tipo de valores debe regresar nuestro intérprete si queremos funciones de primera clase?

Números y funciones.

Funciones de primera clase en FWAE

¿Qué tipo de valores debe regresar nuestro intérprete si queremos funciones de primera clase?

Números y funciones.

Usaremos entonces un nuevo tipo de dato que permita usar estos dos valores como primitivas del lenguaje.

Funciones de primera clase en FWAE

¿Qué tipo de valores debe regresar nuestro intérprete si queremos funciones de primera clase?

Números y funciones.

Usaremos entonces un nuevo tipo de dato que permita usar estos dos valores como primitivas del lenguaje.

Para llevar a cabo la interpretación de funciones, haremos uso de cerraduras (*closures*), los cuales son «una estructura de datos que contiene todo lo que el procedimiento necesita para ser aplicado.». Para los propósitos de este curso, las cerraduras deben almacenar:

Funciones de primera clase en FWAE

¿Qué tipo de valores debe regresar nuestro intérprete si queremos funciones de primera clase?

Números y funciones.

Usaremos entonces un nuevo tipo de dato que permita usar estos dos valores como primitivas del lenguaje.

Para llevar a cabo la interpretación de funciones, haremos uso de cerraduras (*closures*), los cuales son «una estructura de datos que contiene todo lo que el procedimiento necesita para ser aplicado.». Para los propósitos de este curso, las cerraduras deben almacenar:

- Los parámetros formales.

Funciones de primera clase en FWAE

¿Qué tipo de valores debe regresar nuestro intérprete si queremos funciones de primera clase?

Números y funciones.

Usaremos entonces un nuevo tipo de dato que permita usar estos dos valores como primitivas del lenguaje.

Para llevar a cabo la interpretación de funciones, haremos uso de cerraduras (*closures*), los cuales son «una estructura de datos que contiene todo lo que el procedimiento necesita para ser aplicado.». Para los propósitos de este curso, las cerraduras deben almacenar:

- Los parámetros formales.
- El cuerpo de la función.

Funciones de primera clase en FWAE

¿Qué tipo de valores debe regresar nuestro intérprete si queremos funciones de primera clase?

Números y funciones.

Usaremos entonces un nuevo tipo de dato que permita usar estos dos valores como primitivas del lenguaje.

Para llevar a cabo la interpretación de funciones, haremos uso de cerraduras (*closures*), los cuales son «una estructura de datos que contiene todo lo que el procedimiento necesita para ser aplicado.». Para los propósitos de este curso, las cerraduras deben almacenar:

- Los parámetros formales.
- El cuerpo de la función.
- El ambiente donde fue definida la función.

Sustitución diferida

Sustitución diferida

Existen dos maneras de llevar a cabo la sustitución. La primera es hacer sustitución textual, es decir, reemplazar todas las variables ligadas por el valor indicado en la variable de ligado. Esta forma presenta el inconveniente de que su implementación es de orden de $O\left(n^2\right)$ donde n es el número de variables a sustituir. Éste es el método que hemos utilizado hasta ahora.

Sustitución diferida

Existen dos maneras de llevar a cabo la sustitución. La primera es hacer sustitución textual, es decir, reemplazar todas las variables ligadas por el valor indicado en la variable de ligado. Esta forma presenta el inconveniente de que su implementación es de orden de $O\left(n^2\right)$ donde n es el número de variables a sustituir. Éste es el método que hemos utilizado hasta ahora.

La segunda manera consiste en hacer uso de una estructura llamada ambiente, la cual es un repositorio de sustituciones diferidas, es decir, se almacenará en esta estructura de datos cada una de las variables con su valor correspondiente.

Sustitución diferida

Existen dos maneras de llevar a cabo la sustitución. La primera es hacer sustitución textual, es decir, reemplazar todas las variables ligadas por el valor indicado en la variable de ligado. Esta forma presenta el inconveniente de que su implementación es de orden de $O\left(n^2\right)$ donde n es el número de variables a sustituir. Éste es el método que hemos utilizado hasta ahora.

La segunda manera consiste en hacer uso de una estructura llamada ambiente, la cual es un repositorio de sustituciones diferidas, es decir, se almacenará en esta estructura de datos cada una de las variables con su valor correspondiente.

Para lograr implementar esto, se implementarán los ambientes usando una estructura de lista con comportamiento de pila.

Ejemplo

Ejemplo

Veamos cómo se evalúa la siguiente expresión con ambientes:

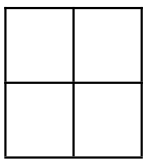
```
{with {x {+ 1 1}}
   {with {y {+ 2 2}}
      {+ x y}}}
```

Ejemplo

Veamos cómo se evalúa la siguiente expresión con ambientes:

```
{with {x {+ 1 1}}
   {with {y {+ 2 2}}
      {+ x y}}}
```

Paso 1: Se empieza la ejecución, inicialmente el ambiente está vacío:



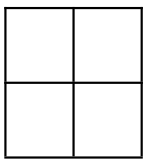
```
{with {x {+ 1 1}} {with {y {+ 2 2} {+ x y}}}
```

Ejemplo

Veamos cómo se evalúa la siguiente expresión con ambientes:

```
{with {x {+ 1 1}}
   {with {y {+ 2 2}}
      {+ x y}}}
```

Paso 1: Se empieza la ejecución, inicialmente el ambiente está vacío:



`{with {x {+ 1 1}} {with {y {+ 2 2} {+ x y}}}`

Paso 2: Se agrega `x` al ambiente, cabe mencionar que primero evaluamos la expresión `(+ 1 1)` que le da valor a la variable `x`.

x	2

`{with {y {+ 2 2}} {+ x y}}`

Ejemplo

Veamos cómo se evalúa la siguiente expresión con ambientes:

```
{with {x {+ 1 1}}
   {with {y {+ 2 2}}
      {+ x y}}}
```

Paso 1: Se empieza la ejecución, inicialmente el ambiente está vacío:

`{with {x {+ 1 1}} {with {y {+ 2 2} {+ x y}}}`

Paso 2: Se agrega `x` al ambiente, cabe mencionar que primero evaluamos la expresión `(+ 1 1)` que le da valor a la variable `x`.

x	2

`{with {y {+ 2 2}} {+ x y}}`

Paso 3: Se agrega `y` al ambiente, debido a la estructura de pila, `y` es ubicada en el tope de la misma.

y	4
x	2

`{+ x y}`

Ejemplo

Ejemplo

Paso 4: Se busca `x` en el ambiente empezando por el tope. Se regresa la primera ocurrencia y se sustituye.

y	4
x	2

```
{+ 2 y}
```

Ejemplo

Paso 4: Se busca `x` en el ambiente empezando por el tope. Se regresa la primera ocurrencia y se sustituye.

y	4
x	2

```
{+ 2 y}
```

Paso 5: Se busca `y` en el ambiente empezando por el tope. Se regresa la primera presencia y se sustituye. Al evaluar esta expresión se llega al resultado de 6.

y	4
x	2

```
{+ 2 4} = 6
```

Ejemplo de cerraduras

Ejemplo de cerraduras

Veamos un ejemplo de una función para obtener su cerradura

```
{with {x 1}
   {with {y 2}
      {with {foo {fun {param} {+ param x}}}
         {foo y}}}}
```

Ejemplo de cerraduras

Veamos un ejemplo de una función para obtener su cerradura

```
{with {x 1}
   {with {y 2}
      {with {foo {fun {param} {+ param x}}}
         {foo y}}}}
```

Parámetros formales:

`param`

Ejemplo de cerraduras

Veamos un ejemplo de una función para obtener su cerradura

```
{with {x 1}
   {with {y 2}
      {with {foo {fun {param} {+ param x}}}
         {foo y}}}}
```

Parámetros formales:

`param`

Cuerpo de la función

`{+ param x}`

Ejemplo de cerraduras

Veamos un ejemplo de una función para obtener su cerradura

```
{with {x 1}
   {with {y 2}
      {with {foo {fun {param} {+ param x}}}
         {foo y}}}}
```

Parámetros formales:

`param`

Cuerpo de la función

`{+ param x}`

Ambiente donde fue definida la función

y	2
x	1

`((x 1) (y 2))`

Nuevos tipos de datos

Nuevos tipos de datos

Necesitamos nuevos tipos de datos para constuir el nuevo intérprete:

Nuevos tipos de datos

Necesitamos nuevos tipos de datos para constuir el nuevo intérprete:

- Un tipo de dato para representar el ambiente.

Nuevos tipos de datos

Necesitamos nuevos tipos de datos para constuir el nuevo intérprete:

- Un tipo de dato para representar el ambiente.
- Un tipo de dato para representar los valores del lenguaje.

Nuevos tipos de datos

Necesitamos nuevos tipos de datos para constuir el nuevo intérprete:

- Un tipo de dato para representar el ambiente.
- Un tipo de dato para representar los valores del lenguaje.

Ambiente

```
(define-type DefrdSub
   [mtSub]
   [aSub (name symbol?) (value FAE-Value?) (ds DefrdSub?)])
```

Nuevos tipos de datos

Necesitamos nuevos tipos de datos para constuir el nuevo intérprete:

- Un tipo de dato para representar el ambiente.
- Un tipo de dato para representar los valores del lenguaje.

Ambiente

```
(define-type DefrdSub
   [mtSub]
   [aSub (name symbol?) (value FAE-Value?) (ds DefrdSub?)])
```

Valores del lenguaje (con cerraduras)

```
(define-type FAE-Value
   [numV (n number?)]
   [closureV (param symbol?) (body FAE?) (ds DefrdSuf?)])
```

En resumen...

En resumen...

Usando estos nuevos tipos de datos, la nueva función interp recibe ahora una expresión y un ambiente. Recordar que al iniciar la ejecución del intérprete el ambiente está vacío.

En resumen...

Usando estos nuevos tipos de datos, la nueva función interp recibe ahora una expresión y un ambiente. Recordar que al iniciar la ejecución del intérprete el ambiente está vacío.

```
(define (interp expr ds)
   (match expr
```

En resumen...

Usando estos nuevos tipos de datos, la nueva función interp recibe ahora una expresión y un ambiente. Recordar que al iniciar la ejecución del intérprete el ambiente está vacío.

```
(define (interp expr ds)
   (match expr
      [(num n) (numV n)]
```

En resumen...

Usando estos nuevos tipos de datos, la nueva función interp recibe ahora una expresión y un ambiente. Recordar que al iniciar la ejecución del intérprete el ambiente está vacío.

```
(define (interp expr ds)
   (match expr
      [(num n) (numV n)]
      [(binop f l r) (binopf f (interp l ds) (interp r ds))]
```

En resumen...

Usando estos nuevos tipos de datos, la nueva función interp recibe ahora una expresión y un ambiente. Recordar que al iniciar la ejecución del intérprete el ambiente está vacío.

```
(define (interp expr ds)
   (match expr
      [(num n) (numV n)]
      [(binop f l r) (binopf f (interp l ds) (interp r ds))]
      [(id v) (lookup v ds)]
      ...))
```

En resumen...

Usando estos nuevos tipos de datos, la nueva función interp recibe ahora una expresión y un ambiente. Recordar que al iniciar la ejecución del intérprete el ambiente está vacío.

```
(define (interp expr ds)
   (match expr
      [(num n) (numV n)]
      [(binop f l r) (binopf f (interp l ds) (interp r ds))]
      [(id v) (lookup v ds)]
      ...))
```

Aplicación de operadores binarios

```
(define (binopf f l r)
   (numV (f (numV-n l) (numV-n r))))
```

En resumen...

Usando estos nuevos tipos de datos, la nueva función interp recibe ahora una expresión y un ambiente. Recordar que al iniciar la ejecución del intérprete el ambiente está vacío.

```
(define (interp expr ds)
   (match expr
      [(num n) (numV n)]
      [(binop f l r) (binopf f (interp l ds) (interp r ds))]
      [(id v) (lookup v ds)]
      ...))
```

Aplicación de operadores binarios

```
(define (binopf f l r)
   (numV (f (numV-n l) (numV-n r))))
```

Función que busca el identificador en el ambiente

```
(define (lookup name ds)
   (match ds
      [(mtSub) (error 'lookup "no binding for identifier")]
      [(aSub bound-name bound-value rest-ds)
         (if (symbol=? bound-name name)
            bound-value
            (lookup name rest-ds))]))
```

Ejercicios

Los ejercicios se encuentran en
`sites.ciencias.unam.mx/ldp171/practicas`

Enviar al correo `manu+ldp@ciencias.unam.mx` **un archivo** `equipo_sesion6.rkt` **con asunto** `[LDP-Sesión 7]` **a más tardar a las 18:59:59 sobre 0.5pts o mañana 29 de septiembre a más tardar a las 17:59:59 sobre 0.25pts.**

Incluir el nombre de los integrantes en el cuerpo del correo y apuntarlos en la hoja en el caso del ejercicio 1.

Sólo pueden entregar aquellos alumnos que aparezcan en la lista de asistencia de la sesión. No es válido apuntar a miembros del equipo que no estén presentes.