Laboratorio de Lenguajes de Programación

Generando código ejecutable - Parte II

Manuel Soto Romero

Universidad Nacional Autónoma de México
Facultad de Ciencias

14 de septiembre de 2016

Objetivos

Objetivos

- Repasar los tipos de análisis que se requieren para obtener código ejecutable.

Objetivos

- Repasar los tipos de análisis que se requieren para obtener código ejecutable.
- Repasar en qué consiste el análisis sintáctico y cómo se relaciona con el análisis semántico.

Objetivos

- Repasar los tipos de análisis que se requieren para obtener código ejecutable.
- Repasar en qué consiste el análisis sintáctico y cómo se relaciona con el análisis semántico.
- Implementar un intérprete sencillo para el lenguaje `FWAE`.

Objetivos

- Repasar los tipos de análisis que se requieren para obtener código ejecutable.
- Repasar en qué consiste el análisis sintáctico y cómo se relaciona con el análisis semántico.
- Implementar un intérprete sencillo para el lenguaje `FWAE`.
- Repasar el algoritmo de sustitución y analizar sus utilidades.

Objetivos

- Repasar los tipos de análisis que se requieren para obtener código ejecutable.
- Repasar en qué consiste el análisis sintáctico y cómo se relaciona con el análisis semántico.
- Implementar un intérprete sencillo para el lenguaje `FWAE`.
- Repasar el algoritmo de sustitución y analizar sus utilidades.
- Crear un archivo ejecutable para el lenguaje `FWAE`.

Recordando...

Recordando...

Recordando las etapas:

Recordando...

Recordando las etapas:

- Escribimos un programa en sintaxis concreta.

Recordando...

Recordando las etapas:

- Escribimos un programa en sintaxis concreta.
- Separamos el programa en lexemas (análisis léxico).

Recordando...

Recordando las etapas:

- Escribimos un programa en sintaxis concreta.
- Separamos el programa en lexemas (análisis léxico).
- Construimos el árbol de sintaxis abstracta (análisis sintáctico).

Recordando...

Recordando las etapas:

- Escribimos un programa en sintaxis concreta.
- Separamos el programa en lexemas (análisis léxico).
- Construimos el árbol de sintaxis abstracta (análisis sintáctico).
- Le damos un significado al árbol de sintaxis abstracta (análisis semántico).

Recordando...

Recordando las etapas:

- Escribimos un programa en sintaxis concreta.
- Separamos el programa en lexemas (análisis léxico).
- Construimos el árbol de sintaxis abstracta (análisis sintáctico).
- Le damos un significado al árbol de sintaxis abstracta (análisis semántico).

Análisis semántico

Análisis semántico

Para darle significado al árbol de sintaxis abstracta, usamos un traductor de lenguaje, ya sea un compilador o un intérprete. En este curso implementamos intérpretes.

Análisis semántico

Para darle significado al árbol de sintaxis abstracta, usamos un traductor de lenguaje, ya sea un compilador o un intérprete. En este curso implementamos intérpretes.

Escribimos entonces una función `(interp exp)` para el lenguaje `FWAE` que iniciamos la sesión pasada.

Análisis semántico

Para darle significado al árbol de sintaxis abstracta, usamos un traductor de lenguaje, ya sea un compilador o un intérprete. En este curso implementamos intérpretes.

Escribimos entonces una función `(interp exp)` para el lenguaje `FWAE` que iniciamos la sesión pasada.

Esta función irá cazando cada uno de los constructores del árbol de sintaxis abstracta generado por el analizador sintáctico.

Análisis semántico

Para darle significado al árbol de sintaxis abstracta, usamos un traductor de lenguaje, ya sea un compilador o un intérprete. En este curso implementamos intérpretes.

Escribimos entonces una función `(interp exp)` para el lenguaje `FWAE` que iniciamos la sesión pasada.

Esta función irá cazando cada uno de los constructores del árbol de sintaxis abstracta generado por el analizador sintáctico.

```
(define (interp exp)
  (match exp
    [(num n) n]
    [(binop f l r) (f (interp l) (interp r))]
    ...))
```

Análisis semántico

Para darle significado al árbol de sintaxis abstracta, usamos un traductor de lenguaje, ya sea un compilador o un intérprete. En este curso implementamos intérpretes.

Escribimos entonces una función `(interp exp)` para el lenguaje `FWAE` que iniciamos la sesión pasada.

Esta función irá cazando cada uno de los constructores del árbol de sintaxis abstracta generado por el analizador sintáctico.

```
(define (interp exp)
  (match exp
    [(num n) n]
    [(binop f l r) (f (interp l) (interp r))]
    ...))
```

La interpretación de números y operaciones binarias es **trivial**.

El algoritmo de sustitución

El algoritmo de sustitución

¿Cómo se evalúa la siguiente expresión?

El algoritmo de sustitución

¿Cómo se evalúa la siguiente expresión?

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

El algoritmo de sustitución

¿Cómo se evalúa la siguiente expresión?

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

La expresión completa se evalúa a `{+ 2 2}`, es decir sustituimos cada aparición de `x` por el valor correspondiente.

El algoritmo de sustitución

¿Cómo se evalúa la siguiente expresión?

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

La expresión completa se evalúa a `{+ 2 2}`, es decir sustituimos cada aparición de `x` por el valor correspondiente.

Necesitamos entonces, un algoritmo que nos permita hacer estas sustituciones. Definimos esta función como `(subst expr sub-id val)`, es decir necesitamos una expresión donde sustituir, un valor a sustituir el valor con el que se realizará.

Sustitución de números y operaciones binarias

Sustitución de números y operaciones binarias

```
(define (subst expr sub-id val)
   (match expr
      [(num n) expr]
      [(binop f l r) (binop f
                  (subst l sub-id val)
                  (subst r sub-id val))]
      ...))
```

Sustitución de números y operaciones binarias

```
(define (subst expr sub-id val)
   (match expr
      [(num n) expr]
      [(binop f l r) (binop f
                  (subst l sub-id val)
                  (subst r sub-id val))]
      ...))
```

- El intentar sustituir un número no tiene efecto alguno, pues no hay identificador para sustituir.

Sustitución de números y operaciones binarias

```
(define (subst expr sub-id val)
   (match expr
      [(num n) expr]
      [(binop f l r) (binop f
                  (subst l sub-id val)
                  (subst r sub-id val))]
      ...))
```

- El intentar sustituir un número no tiene efecto alguno, pues no hay identificador para sustituir.

- Sustituir una operación binaria, es sustituir recursivamente en el lado izquierdo y el lado derecho respectivamente.

Sustitución de asignaciones locales

Sustitución de asignaciones locales

Para `with` tenemos que considerar dos casos:

Sustitución de asignaciones locales

Para `with` tenemos que considerar dos casos:

```
[(with bound-id named-expr bound-body)
   (if (symbol=?  bound-id sub-id)
   ...]
```

Sustitución de asignaciones locales

Para `with` tenemos que considerar dos casos:

```
[(with bound-id named-expr bound-body)
   (if (symbol=?  bound-id sub-id)
   ...]
```

Si el identificador de la expresión `with` es igual al del valor a sustituir, simplemente sustituimos en la expresión asociada al identificador, pues el alcance del cuerpo del with hace referencia al identificador y no podemos cambiar dicho valor:

Sustitución de asignaciones locales

Para `with` tenemos que considerar dos casos:

```
[(with bound-id named-expr bound-body)
   (if (symbol=?  bound-id sub-id)
   ...]
```

Si el identificador de la expresión `with` es igual al del valor a sustituir, simplemente sustituimos en la expresión asociada al identificador, pues el alcance del cuerpo del with hace referencia al identificador y no podemos cambiar dicho valor:

```
(with bound-id
   (subst named-expr sub-id val)
   bound-body)
```

Sustitución de asignaciones locales

Sustitución de asignaciones locales

Si el identificador de la expresión `with` es distinto al del valor a sustituir, sustituimos la expresión asociada al indentificador y al cuerpo, pues esto no afecta el alcance.

Sustitución de asignaciones locales

Si el identificador de la expresión `with` es distinto al del valor a sustituir, sustituimos la expresión asociada al indentificador y al cuerpo, pues esto no afecta el alcance.

```
(with bound-id
   (subst named-expr sub-id val)
   (subst bound-body sub-id val))
```

Sustitución de asignaciones locales

Si el identificador de la expresión `with` es distinto al del valor a sustituir, sustituimos la expresión asociada al indentificador y al cuerpo, pues esto no afecta el alcance.

```
(with bound-id
   (subst named-expr sub-id val)
   (subst bound-body sub-id val))
```

¿En qué caso cae esta llamada?

```
(subst (with 'x (num 2) (binop + (num 3) (id 'x))) 'x (num 3))
```

Sustitución de asignaciones locales

Si el identificador de la expresión `with` es distinto al del valor a sustituir, sustituimos la expresión asociada al indentificador y al cuerpo, pues esto no afecta el alcance.

```
(with bound-id
   (subst named-expr sub-id val)
   (subst bound-body sub-id val))
```

¿En qué caso cae esta llamada?

```
(subst (with 'x (num 2) (binop + (num 3) (id 'x))) 'x (num 3))
```

En el primero.

Sustitución de asignaciones locales

Si el identificador de la expresión `with` es distinto al del valor a sustituir, sustituimos la expresión asociada al indentificador y al cuerpo, pues esto no afecta el alcance.

```
(with bound-id
   (subst named-expr sub-id val)
   (subst bound-body sub-id val))
```

¿En qué caso cae esta llamada?

```
(subst (with 'x (num 2) (binop + (num 3) (id 'x))) 'x (num 3))
```

En el primero.

¿En qué caso cae esta llamada?

```
(subst (with 'x (num 2) (binop + (id 'y) (id 'x))) 'y (num 3))
```

Sustitución de asignaciones locales

Si el identificador de la expresión `with` es distinto al del valor a sustituir, sustituimos la expresión asociada al indentificador y al cuerpo, pues esto no afecta el alcance.

```
(with bound-id
   (subst named-expr sub-id val)
   (subst bound-body sub-id val))
```

¿En qué caso cae esta llamada?

```
(subst (with 'x (num 2) (binop + (num 3) (id 'x))) 'x (num 3))
```

En el primero.

¿En qué caso cae esta llamada?

```
(subst (with 'x (num 2) (binop + (id 'y) (id 'x))) 'y (num 3))
```

En el segundo.

Interpretando asignaciones locales

Interpretando asignaciones locales

Interpretamos el cuerpo con la sustitución del identificador correspondiente.

Interpretando asignaciones locales

Interpretamos el cuerpo con la sustitución del identificador correspondiente.

```
[(with bound-id named-expr bound-body)
   (interp (subst bound-body bound-id (num (interp named-expr))))]
```

Interpretando asignaciones locales

Interpretamos el cuerpo con la sustitución del identificador correspondiente.

```
[(with bound-id named-expr bound-body)
   (interp (subst bound-body bound-id (num (interp named-expr))))]
```

Interpretando funciones

Interpretando funciones

En esta versión del intérprete, no regresamos ningún resultado al interpretar una función. Una función es una función y punto.

Interpretando funciones

En esta versión del intérprete, no regresamos ningún resultado al interpretar una función. Una función es una función y punto.

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

Generando archivo ejecutable

Generando archivo ejecutable

Una vez que la función que realiza la interpretación es correcta. Escribimos un programa que interactúe con el usuario mediante `read`. Por ejemplo:

Generando archivo ejecutable

Una vez que la función que realiza la interpretación es correcta. Escribimos un programa que interactúe con el usuario mediante `read`. Por ejemplo:

```
(define (ejecuta)
   (begin
      (display "(λ) ")
      (define x (read))
      (if (equal?  x '{exit})
         (display "")
         (begin
            (display (interp (parse x)))
            (display "\n")
            (ejecuta)))))
;; Llamada a función encargada de iniciar la ejecución del
interprete
(display "Bienvenido a FWAE v1.0.\n")
(ejecuta)
```

Una vez definida la función anterior...

Una vez definida la función anterior...

- En el menú «Racket» elegimos «Crear ejecutable».

Una vez definida la función anterior...

- En el menú «Racket» elegimos «Crear ejecutable».
- En tipo elegimos «Stand-alone».

Una vez definida la función anterior...

- En el menú «Racket» elegimos «Crear ejecutable».
- En tipo elegimos «Stand-alone».
- En base elegimos «Racket».

Una vez definida la función anterior...

- En el menú «Racket» elegimos «Crear ejecutable».
- En tipo elegimos «Stand-alone».
- En base elegimos «Racket».
- Presionamos «Crear».

Una vez definida la función anterior...

- En el menú «Racket» elegimos «Crear ejecutable».
- En tipo elegimos «Stand-alone».
- En base elegimos «Racket».
- Presionamos «Crear».
- Para ejecutar el archivo creado simplemente escribimos ./NombreArchivo.

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 6]` **a más tardar a las 18:59:59.**

Incluir el nombre de los integrantes en el cuerpo del correo.

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.