Universidad Nacional Autónoma de México
Facultad de Ciencias
Lenguajes de Programación

Nota de clase 2: Generación de código ejecutable

Karla Ramírez Pulido
karla@ciencias.unam.mx

Manuel Soto Romero
manu@ciencias.unam.mx

18 de marzo de 2018

Descripción

Para que una computadora ejecute las órdenes escritas en un código fuente, éste pasa por una serie de análisis que permiten obtener código ejecutable. En estas notas se construye un pequeño intérprete para el lenguaje de programación `WAE` y se discuten los pasos a seguir en cada uno de los tipos de análisis para obtener un programa ejecutable.

Índice general

Índice de figuras

Índice de códigos

2.1 Sintaxis concreta

La *sintaxis concreta* se refuere a cómo se escriben las expresiones de un determinado lenguaje de programación. Estas reglas de escritura son de alto nivel, es decir, son, en un principio, más comprensibles para el usuario final del lenguaje y no tanto para la arquitectura o el lenguaje de la máquina.

Para describir las reglas de escritura, lo usual es hacerlo mediante la llamada Forma Extendida de Backus Naur (EBNF[1] por sus siglas en inglés). Se describen, a continuación Las reglas de escritura para las expresiones del lenguaje de programación `WAE` usando esta notación:

```
<expr> ::= <id>
         | <num>
         | {<binop> <expr> <expr>}
         | {with {<id> <expr>} <expr}

<id> := a |...|z|A|...|Z|aa|...|zz|...
        (Cualquier conmbinación de carácteres)

<num> := ...|-2|-1|0|1|2|...

<binop> := +|-|*|/
```

Ejemplo 2.1 Algunas expresiones del lenguaje `WAE`:

```
foo      1729      {+ 17 29}      {- {+ 17 29} {* 18 35}}

{/ {+ a b} 2}     {with {a 2}       {with {a 2}
                     {with {b 3}      {with {b 3}
                       {+ a b}}}        {with {c {with {d 4} {* d d}}}
                                            {+ a {* b c}}}}}
```

Se aprecia en el Ejemplo 2.1 que las expresiones del lenguaje antes definido, usan notación prefija y llaves. Una sintaxis muy parecida a la de Racket y otros lenguajes descendientes de Lisp.

2.2 Análisis léxico

Una expresión en sintaxis concreta, es en sí una cadena (secuencia de caracteres). El primer análisis por el que pasa un código consiste en tomar estas cadenas y separarlas en lexemas[2].

[1] Extended Backus Naur Form, llamada así en honor a su inventor John Backus y a Peter Naur, quien la modificó para utilizarla en las especificaciones del lenguaje ALGOL.

[2] La RAE define lexema como la unidad mínima con significado léxico que no presenta morfemas gramaticales.

Por ejemplo, en la expresión `{+ 17 29}`, los lexemas involucrados son las llaves, el operador de suma y los números 17 y 29.

El programa encargado de hacer esta separación de lexemas es llamado *analizador léxico*[3]. Existen versiones más avanzadas de este tipo de analizadores que regresan una lista con parejas de la forma `(tipo, valor)`. Por ejemplo, la expresión:

```
a = a + b
```

se separa en lexemas, como: `[(id,a),(asig,=),(id,a),(op,+),(id,b)]`.

Para los fines de estas notas, basta con regresar una lista de lexemas sin conocer su papel sintáctico (tipo). Las formas de impleentación de analizadores léxicos suelen estudiarse a fondo en cursos de compiladores haciendo uso de la teoría de autómatas y lenguajes formales.

Notación `quote`

Para ahorrarse el trabajo de obtener los lexemas de una expresión, en Racket, se hace uso de una primitiva muy usada pos los lenguajes descendientes de Lisp llamada `quote`. Cuando se aplica `quote` a una expresión, lo que se está diciendo es:

Tómalo literal, no lo intérpretes

Todo lo que aparezca después de `quote` no se interpreta y toma su valor como tal.

Ejemplo 2.2 Una expresión sin `quote` y con `quote`:

```
> {+ 17 29}
46
> (quote {+ 17 29})
'{+ 17 29}
> '{+ 17 29}
'{+ 17 29}
```

En el ejemplo anterior, se toma tal cual la expresión recibida, no se está interpretando y de hecho, en este caso, se obtiene una lista con los lexemas de la expresión.

[3] *Scanner* o *lexer* en inglés

Observación 2.1 Es equivalente usar la función `quote` que agregar el símbolo `'` al inicio de una expresión.

Un analizador léxico toma una cadena y la descompone en lexemas. De esta forma, se necesita un mecanismo que tome una cadena y aplique `quote` a ésta. Este mecanismo existe en Racket y se llama `read`.

Ejemplo 2.3 Una expresión dada por el usuario con la aplicación de `quote` a partir de la llamada a `read`.

```
> (read)
{+ 1 2}
'{+ 1 2}
```

Observación 2.2 Al escribir `(read)` en el área de interacciones de DrRacket, aparece un campo dónde el usuario puede ingresar una cadena, `read` toma esta cadena y regresa la lista de lexemas correspondiente. La primitiva `quote`, siempre regresa: un número, un símbolo o una lista.

2.3 Análisis sintáctico

Es natural pensar que ahora que se tiene una expresión separada en lexemas, se puede evaluar y mostrar los resultados, sin embargo, la sintaxis concreta con que se definen las expresiones es clara para el usuario, pero no lo es tanto para la computadora, pues se presta a ambigüedades.

Por ejemplo, las siguientes expresiones:

```
1 + 2
+ 1 2
1 2 +
```

Es claro para el usuario, es que están denotando lo mismo: *sumar uno con dos*. Sin embargo, para la computadora, las tres expresiones son distintas. Se necesita entonces, una representación intermedia que permita abstraer este tipo de situaciones.

La abstracción es un principio por el cual se ignora toda aquella información que no es importante en un contexto particular. En este caso, se necesita omitir la notación de las expresiones (prefija, infija o postfija). De esta forma, la representación intermedia que se puede adoptar son los *Árboles de Sintaxis Abstracta* (ASA).

La Figura 1 muestra el árbol de sintaxis abstracta para las expresiones anteriores. La notación que se usa para representar a los ASA es en formato de lista, además de añadir un poco más de información en cada nodo, para saber qué se está almacenando en cada uno. Por ejemplo:

```
(add (num 1) (num 2))
```

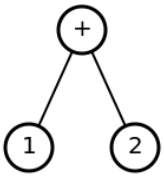

Figura 1: *Árbol de sintaxis abstracta*

ASA de expresiones de WAE

Para representar un ASA en Racket se usa la primitiva `define-type` dada por el dialecto `plai`. Cada constructor representa la raíz del ASA y sus parámetros son los hijos.

Los ASA para WAE se definen cómo:

```
;; TDA para representar los árboles de sintaxis abstracta del
;; lenguaje WAE.
(define-type WAE
    [id    (i symbol?)]
    [num   (n number?)]
    [binop (op procedure?) (izq WAE?) (der WAE?)]
    [with  (id symbol?) (value WAE?) (body WAE?)])
```

Código 1: *TDA para representar los lenguaje de sintaxis abstracta del lenguaje WAE*

Ejemplo 3.3 Algunos ejemplos de ASA para expresiones del lenguaje WAE:

```
Sintaxis concreta: foo
Sintaxis abstracta: (id foo)

Sintaxis concreta: 1729
Sintaxis abstracta: (num 1729)

Sintaxis concreta: {+ 17 29}
Sintaxis abstracta: (binop + (num 17) (num 29))

Sintaxis concreta: {- {+ 17 29} {* 18 35}}
Sintaxis abstracta: (binop -
                          (binop + (num 17) (num 29))
                          (binop * (num 18) (num 25)))

Sintaxis concreta: {/ {+ a b} 2}
Sintaxis abstracta: (binop /
                          (binop + (id 'a) (id 'b)) (num 2))
```

```
Sintaxis concreta: {with {a 2}
                         {with {b 3}
                            {+ a b}}}
Sintaxis abstracta: (with 'a
                          (num 2)
                          (with 'b
                                (num 3)
                                (binop + (id 'a) (id 'b))))


Sintaxis concreta: {with {a 2}
                         {with {b 3}
                            {with {c {with {d 4} {* d d}}}
                               {+ a {* b c}}}}}
Sintaxis abstracta: (with 'a
                          (num 2)
                          (with 'b
                                (num 3)
                                (with 'c
                                      (with 'd (binop * (id 'd) (id 'd)))
                                      (binop +
                                             (id 'a)
                                             (binop * (id 'b) (id 'c))))))
```

Analizador sintáctico para WAE

El análisis sintáctico, consiste en tomar una expresión en sintaxis concreta y pasarla a una representación intermedia que sea comprensible para la computadora. En este caso, un árbol de sintaxis abstracta (un mapeo). En esta fase se detectan errores de sintaxis, es decir, si las expresiones están bien formadas sintácticamente.

Estos analizadores reciben el nombre de analizador sintáctico[4]. Se escribe una función `parse` que realiza el análisis sintáctico. Debido a que el analizador léxico regresa números, símbolos o listas, se usa en gran medida la técnica de apareamiento de patrones mediante la primitiva `match`[5], para construir el ASA correspondiente.

```
;; Función que toma una expresión en sintaxis concreta y regresa
;; el árbol de sintaxis abstracta correspondiente.
;; parse: symbol -> WAE
(define (parse sexp)
   (match sexp
      [(? symbol?) (i sexp)]
      [(? number?) (num sexp)]
```

[4] *Parser.*

[5] También puede usarse `type-case`.

```
        [(list 'with (list id value ) body)
           (with id (parse value) (parse body))]
        [(list op ziq der)
           (binop (elige op) (parse l) (parse r))]))

;; [Auxiliar]. Función que hace un mapeo entre los operadores en
;; sintaxis concreta y los operadores de Racket. Esto con el fin de
;; aplicar la operación más adelante.
;; elige: symbol -> procedure
(define (elige sexp)
   (match sexp
      ['+ +]
      ['- -]
      ['* *]
      ['/ /]))
```

Código 2: *Analizador sintáctico para WAE*

2.4 Análisis semántico

Una vez que se construye el ASA correspondiente, se procede a evaluarlo. La semántica de un lenguaje de programación se refiere al comportamiento asociado a la sintaxis, por ejemplo, cuando se escribe `{+ 17 29}` en sintaxis concreta, el intérprete o compilador detecta que el usuario ingresó una suma con dos operandos y darle el comportamiento (significado) correspondiente, regresando el valor 46, que es el esperado por el usuario[6].

A la fase encargada de evaluar las expresiones se le conoce como *análisis semántico*. De esta forma, se define la función `(interp expr)` que recibe un árbol de sintaxis abstracta y regresa la evaluación correspondiente. Un intérprete para el lenguaje de programación `WAE`, puede evaluar expresiones como sigue:

- Un identificador libre debería reportar un error, por ejemplo:

  ```
  > foo
  error: Identificador libre
  ```

- La evaluación de un número no es otra cosa, más que él mismo, por ejemplo:

  ```
  > 1729
  1729
  ```

- Evaluar una operación binaria, es aplicar la función (de Racket) a los dos parámetros de la función, por ejemplo:

[6]Los lenguajes de programación no siempre regresan los valores esperados por el usuario, esto depende de la visión del diseñador del lenguaje.

```
> {+ 17 {* 29 18}}
539
```

- Una asignación local permite introducir variables con alcance restringido. Generan expresiones más eficientes con respecto a la evaluación, pues el valor ligado a una variable se calcula una única vez. Para evaluar este tipo de expresiones se debe sustituir el valor de los identificadores en el cuerpo de la expresión y regresar la evaluación del mismo, por ejemplo[7]:

```
> {with {a 2} {+ a a}}
> {with {a 2} {+ 2 2}}
> {+ 2 2}
4
```

Algoritmo de sustitución textual

Se necesita de otra función auxiliar que permita sustituir el valor delr identificador correspondiente dentro de una expresión de `WAE`. Es necesaria para evaluar las expresiones `with`.

En cursos, el de como Estructuras Discretas o el de Lógica Computacional se estudia el concepto de *sustitución textual* y se presentan algunas definiciones que son de utilidad.

Definición 2.1 (Sustitución textual) Supóngase que se tienen dos expresiones E y R, y sea x un identificador (usualmente, pero no siempre, presente en E). Se usa la notación $E[x:=R]$ para denotar la expresión que es la misma que E, pero dónde cada presencia (ocurrencia) de x en la expresión E ha sido sustituida por la expresión R. Se llama **sustitución textual** al acto de sustituir todas las presencias de x en E por R.

Definición 2.2 (Instancia de ligado) La **instancia de ligado** de un identificador es la instancia de un identificador que da a éste su valor. En `WAE` la posición del id de un `with` es la única instancia de ligado.

Definición 2.3 (Alcance) El **alcance** de una instancia de ligado es la región del programa en la cual las instancias de un identificador se refieren al valor ligado de alguna instancia de ligado.

Definición 2.4 (Instancia ligada) Un identificador está **ligado** si está contenido en el alcance de una instancia de ligado por su nombre.

Definición 2.5 (Instancia libre) Un identificador que no está contenido en el alcance de cualquier instancia de ligado se dice que está **libre.**

Por ejemplo, en la expresión:

[7] Se muestra la evaluación paso a paso a manera de ejemplo, por lo general los intérpretes omiten estos pasos.

```
{with {a 2}
   {with {b 3}
      {+ a {* b c}}}}
```

- Las instancias de ligado son: `a` y `b`.
- El alcance de las instancias `a` y `b` es el cuerpo del `with` correspondiente.
- En la expresión `{+ a {* b c}}` los identificadores `a` y `b` son instancias ligadas y el identificador `c` es una instancia libre.

La función `(subst expr sub-id val)` realiza la sustitución textual *expr[sub-id:=val]* y se define como sigue:

```
;; Función que toma una expresión, un identificador y un valor
;; y regresa la sustitución en la expresión de los identificadores
;; con el valor correspondiente.
;; subst: WAE symbol WAE -> WAE
(define (subst expr sub-id val)
   (match expr
      [(id i) (if (symbol=? i sub-id) val expr)]
      [(num n) expr]
      [(binop op izq der)
         (binop op (subst izq sub-id val) (subst der sub-id val))]
      [(with id value body)
         (if (symbol=? id sub-id)
             (with
                id
                (subst value sub-id val)
                body)
             (with
                id
                (subst value sub-id val)
                (subst body sub-id val)))]))
```

Código 3: *Implementación del algoritmo de sustitución*

Observación 2.3 Al sustituir en una expresión `with`:

- Si el identificador de la expresión es igual al del valor a sustituir, simplemente se sustituye en la expresión asociada al identificador, pues el alcance del cuerpo del `with` hace referencia al identificador y no se puede cambiar dicho nombre.
- Si el identificador de la expresión es distinto al del valor a sustituir, se sustituye en la expresión asociada al identificador y en el cuerpo, pues esto no afecta al alcance.

Analizador semántico para WAE

Finalmente, se procede a definir la función `interp`:

```
;; Función que toma un árbol de sintaxis abstracta y regresa su
;; evaluación.
;; interp: WAE -> number
(define (interp expr)
   (match expr
      [(id i) (error 'interp "Identificador libre")]
      [(num n) n]
      [(binop op izq der)
          (f (interp izq) (interp der))]
      [(with id value body)
          (interp (subst body id value))]))
```

Código 4: *Analizador semántico para WAE*

Se aprecia cómo para interpretar un `with` se hace uso del algoritmo de sustitución textual.

2.5 Optimizaciones

El objetivo de una optimización de código es descubrir, en tiempo de compilación, información acerca del comportamiento del programa en tiempo de ejecucióny usar esta información para mejorar el código ejecutable generado por el intérprete o compilador. Mejorar el código ejecutable puede hacerse en varios sentidos y formas. El objetivo más común de una optimización es hacer que el código ejecutable sea más rápido.

No se profundiza en la implementación de estas optimizaciones, pero se hace mención de varias técnicas a lo largo de las siguientes notas, por ejemplo la técnica de recursión de cola. Por lo pronto basta con que el lector sepa que éstas existen.

3.6 Referencias

[1] Rodrigo Ruiz Murgía, *Manual de prácticas para la asignatura de Lenguajes de Programación*, Reporte de actividad docente, Facultad de Ciencias, 2016.

[2] Shriram Krishnamurthi, *Programming Languages: Application and Interpretation*, Primera edición, Brown University, 2007.

[3] Favio E. Miranda Perea, Elisa Viso Gurovich, *Matemáticas Discretas,* Primera edición, Las prensas de Ciencias, 2009.

[4] Keith D. Cooper, Linda Torczon, *Engineering a Compiler,* Segunda edición, Morgan Kaufman, 2012.

Nota: Esta es una versión preliminar de las notas de laboratorio y se encuentran en constante actualización por lo que agradecemos que en caso de detectar algún error o si se desea hacer alguna observación se envíe un correo electrónico a las direcciones `manu@ciencias.unam.mx` y `karla@ciencias.unam.mx` con el asunto `Nota 2: Lenguajes de Programación`.