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

6 de septiembre de 2017
Semestre 2018-1

Descripción

Antes de que una computadora ejecute las órdenes dadas por un programador a partir de un código fuente, éste pasa por una serie de fases que permiten obtener un código ejecutable. En estas notas se construirá un pequeño intérprete para el lenguaje `WAE` y se discutirán 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

3.1 Sintaxis concreta

Nos referimos con *sintaxis concreta* a cómo se escriben las expresiones de un determinado lenguaje. Estas reglas de escritura son de un alto nivel, es decir, tratan de ser más comprensibles para el usuario final del lenguaje y no tanto para la arquitectura o el lenguaje de la máquina.

Para describir estas reglas de escritura, lo usual es hacerlo mediante la llamada Forma Extendida de Backus Naur (EBNF[1] por sus siglas en inglés).

Lo primero que se hará, es describir las reglas de escritura para las expresiones del lenguaje `WAE` usando EBNF:

```
<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 3.1 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}}}}}
```

•

En estos ejemplos podemos notar que las expresiones del lenguaje definido, usan notación prefija y paréntesis, una sintaxis muy parecida a la de Racket y otros lenguajes descendientes de Lisp.

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

3.2 Análisis léxico

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

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

El programa encargado de hacer esta separación es conocido como *analizador léxico* (*scanner* o *lexer* en inglés). Existen versiones más avanzadas de estos analizadores que en lugar de regresar una lista con los lexemas, regresan una lista con parejas de la forma `(tipo, valor)`.

Por ejemplo, la expresión:

```
a = a + b
```

se convierte en `[(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 implementar analizadores léxicos suelen estudiarse a fondo en cursos de estudio 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, se hará uso de un primitiva típica de 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

En pocas palabras, todo lo que aparezca después de `quote` no se interpretará y tomará su valor como tal.

Ejemplo 3.2 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 literal la expresión recibida, no se está interpretando y de hecho, en este caso, se regresa una lista con los lexemas de la expresión.

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

Observación 3.1 Da lo mismo usar la función `quote` que agregar el símbolo `'` al inicio de una expresión. Ambas formas son equivalentes.

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

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

•

Observación 3.2 Al escribir `(read)` en el área de interacciones de DrRacket, aparece un campo dónde el usuario puede ingresar una cadena, `read` la toma y regresa la lista con los lexemas correspondientes. Es importante mencionar, que `quote`, siempre regresará: un número, un símbolo o una lista.

3.3 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 mucho para la computadora, pues se presta a ambigüedades.

Por ejemplo, las siguientes expresiones:

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

Para el usuario, es claro que las tres expresiones 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 una determinada situación. En este caso, se quiere omitir, por ejemplo, si las expresiones se escriben notación prefija, infija o postfija. De esta manera, la representación intermedia que adoptaremos serán *Árboles de Sintaxis Abstracta* (ASA).

La Figura 1 muestra el árbol de sintaxis abstracta para las tres expresiones anteriores. La notación que se usará en estas notas para representar a los ASA será 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))
```

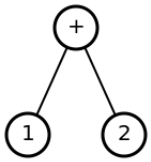

Figura 1: *Árbol de sintaxis abstracta*

ASA de expresiones de WAE

Para representar los ASA de los lenguajes que se revisarán a lo largo de estas notas, se usará la primitiva `define-type` que provee el dialecto `plai` de Racket. De esta forma, cada constructor representará la raíz del ASA y sus parámetros serán los hijos.

Los ASA para WAE se definen cómo:

```
;; TDA para representar los árboles de sintaxis abstracta del
;; lenguaje WAE.
(define-type WAE
   [id (id symbol?)]
   [num (n number?)]
   [binop (f procedure?) (izq WAE?) (der WAE?)]
   [with (name symbol?) (named-expr 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 de tomar una expresión en sintaxis concreta y construir el árbol de sintaxis abstracta correspondiente (un mapeo). En esta fase se detectan errores de sintaxis, es decir, si las expresiones están bien formadas sintácticamente.

Típicamente los analizadores reciben el nombre de analizador sintáctico (*parser*), así se escribirá una función parse que realice el análisis sintáctico. Debido a que el analizador léxico regresa números, símbolos o listas, se usará en gran medida la técnica de apareamiento de patrones (*pattern matching)* usando la primitiva `match`, 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?) (id sexp)]
      [(? number?) (num sexp)]
      [(list 'with (list id val) body)
         (with id (parse val) (parse body))]
```

```
      [(list op l r)
         (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*

3.4 Análisis semántico

Una vez que se tiene construido el ASA correspondiente, se procede a evaluarlo. Recordando que la semántica de un lenguaje de programación se refiere al comportamiento asociado a la sintaxis, por ejemplo, cuando se escribe $\{+\ 17\ 29\}$ en sintaxis concrete, el intérprete o compilador debe detectar que el usuario ingresó una suma con dos operandos y darle el comportamiento (significado) correspondiente, regresando el valor 46, que es el valor esperado por el usuario[3].

A la fase encargada de evaluar las expresiones se le conoce como análisis semántico 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 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:

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

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

- Finalmente, para evaluar una asignación local, tenemos que sustituir el valor de los identificadores en el cuerpo de la expresión y regresar la evaluación del mismo, por ejemplo[4]:

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

Algoritmo de sustitución textual

Antes de implementar la función `interp`, se necesita de otra función auxiliar que permita sustituir el valor de un identificador correspondiente dentro de una expresión de WAE pues se necesitará para interpretar las expresiones `with`.

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

Definición 3.1 (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 3.2 (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 3.3 (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 3.4 (Instancia ligada) Un identificador está **ligado** si está contenido en el alcance de una instancia de ligado por su nombre.

Definición 3.5 (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:

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

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

- 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.

De esta forma, la función `(subst expr sub-id val)` que realiza la sustitución textual *expr[sub-id:=val]* 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 f izq der)
         (binop f (subst izq sub-id val) (subst der sub-id val))]
      [(with bound-id named-expr body)
         (if (symbol=? bound-id sub-id)
             (with
                bound-id
                (subst named-expr sub-id val)
                bound-body)
             (with
                bound-id
                (subst named-expr sub-id val)
                (subst bound-body sub-id val)))]))
```

Código 3: *Implementación del algoritmo de sustitución*

Observación 3.3 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 podemos 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 el alcance.

Analizador semántico para WAE

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

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

Código 4: *Analizador semántico para WAE*

3.5 Optimizaciones

El objetivo de las optimizaciones de código es descubrir, en tiempo de compilación, información acerca del comportamiento en tiempo de ejecución del programa y 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 se ejecute más rápido.

No se profundizará en la implementación de estas optimizaciones, pero se hará mención de varias técnicas a lo largo de las notas siguientes, por ejemplo la técnica de recursión de cola. Por lo pronto basta con que el lector sabe 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 L3: Lenguajes de Programación`.