Compiladores 2023-2
Facultad de Ciencias UNAM
Práctica 4: Jelly a Nanopass.

Lourdes del Carmen Gonzáles Huesca Juan Alfonso Garduño Solís
Braulio Aaron Santiago Carrillo

15 de marzo
Fecha de Entrega: 31 de marzo

Prelimiares

El objetivo de esta práctica definir un lenguaje en Nanopass que sea capaz de reconocer la representación intermedia en la que nos encontramos trabajando. El parser definido en la tercer práctica provee un árbol de sintáxis abstracta que si bien sirve para corroborar que un programa sea sintácticamente congruente con las especificaciones de Jelly, está almacenando información de más y por eso antes de comenzar con la definicion del lenguaje de nanopass es prudente aligerarlo.
Comenzaremos entonces con una función que además de pasar todo a notación prefija, como lo requiere nanopass, quite información que es irrelevante después de la etapa de parseo. Por ejemplo para el método gdc que es el siguiente:

```
gcd(a:int, b:int):int{
    while (a != 0){
        if (a < b)
            b = b - a
        else
            a = a - b
    }
    return b
}
```

el parser de la práctica pasada generaría un árbol parecido al siguiente:

```
(metodo (id 'gcd)
        (list (decl (id 'a) 'INT) (decl (id 'b) 'INT) (decl (id 'x) 'INT))
        'INT
        (list
            (while-exp (bin-exp '!= (id 'a) (num 0))
                (if-exp  (bin-exp '< (id 'a)
                         (id 'b))
                         (bin-exp '= (id 'b) (bin-exp '- (id 'b) (id 'a)))
                         (bin-exp '= (id 'a) (bin-exp '- (id 'a) (id 'b)))))
            (return (id 'b))))
```

esta representación aún puede tener azúcar sintáctica que no retiramos en la práctica pasada o contenedores de más que pueden complicar la evaluación y el análisis de casos, por eso buscamos convertirlo a algo como esto:

```
(gcd [(a int) (b int) (x int)] int
    {(while (!= a 0)
        {(if (< a b)
             (= b (- b a))
             (= a (- a b)))}
     )
     (return b)})
```

Este árbol ya no es tan sintacticamente riguroso y tiene sentido preferirlo en este momento de la compilación porque ya terminamos el proceso de analisis sintáctico.

Implementación.

De manera similar a como especificamos la gramática en el parser de la practica pasada, vamos a usar pattern-matching para especificar el comportamiento de la función

```
(define (->nanopass e)
    (match e
   ;[patron acción]
    [(num n) (number->string n)]))
```

y como te habrás dado cuenta por el ejemplo vamos a dar el resultado como una cadena, esto nos va a dar completa libertad en como construir nuestra nueva representación intermedia, por ejemplo para una expresión binaria:

```
(define (->nanopass e)
    (match e
   ;[patron acción]
    [(num n) (number->string n)]
    [(bin-exp '+ exp1 exp2) (string-append
                             "(+ " (->nanopass exp1) (->nanopass exp2)")")]))
```

Los casos dependerán de como hayas implementado tu parser.

Nanopass

Ahora, nanopass es una herramienta que permite desarrollar compiladores compuestos de múltiples pasos pequeños (por eso el nombre), cada uno de estos pasos tiene un propósito específico que modifíca lenguaje fuente en una serie de lenguajes intermedios bien definidos.
El fin es simplificar y comprender mejor cada paso del compilador para modularizarlo de tal forma que si se quieren agregar nuevas fases sea sencillo y no implique una reestructuración del proyecto.
Para el último ejercicio de esta práctica tienes que definir un lenguaje que reconozca la salida del ejercicio anterior, el cascarón de la definición de un lenguaje básico se ve así:

```
(define-language nombre
    (terminals
        (simbolo-terminal (meta-var))
        ...)
    (variable-no-terminal (meta-var ...)
     producciones ...))
```

Dónde:

- `nombre` es el identificador del lenguaje.
- `simbolo-terminal` es el nombre que revibe la una expresión terminal.
- `meta-var` en ambos casos es un identificador para referirnos a la `variable-no-terminal` o `simbolo-terminal` al que está asociado.
- `variable-no-terminal` es un identificador para un símbolo no termial de las formas aceptadas por las producciones que se definen en su alcance (igual en concepto a una meta-variable).
- `producciones` es una *s-expression* que representa un patrón válido del lenguaje que estamos definiendo.

Por ejemplo, en la definición del siguiente lenguaje

```
(define-language ejemplo
    (terminals
        (constante (c))
        (primitivo (pr)))
    (Expr (e)
     c
     pr
     (pr c1 c2)))
```

- `ejemplo` es el nombre del lenguaje, se utiliza para por ejemplo, definir el parser `(define-parser parser-ejemplo ejemplo)`
- `c` es la meta-variable con la que nos referimos a una constante en las producciones.
- `pr` es la meta-variable para referirnos a una operación primitiva en las producciones
- `Expr` es el equivalente a un símbolo no terminal en una gramática, a partir del cual podemos derivar cualquiera de sus producciones:

  ```
  Expr -> c | pr | (pr c c)
  ```

- `e` es una meta-variable para referirnos a Expr dentro de otras producciones.

Y finalmente para que todo esto funcione necesitamos predicados para saber si algo es una constante o un primitivo, por eso es necesario la definicion de las siguientes funciones:

```
(define (primitivo? p) (memq p '(+)))
(define (constante? c) (number? c))
```

Esto es lo necesario para que definas tu propio lenguaje

Ejercicios.

- ***(5 puntos)*** Implementa una función `->nanopass` que reciba el árbol que se obtiene del parser de tu práctica 3 para generar uno más ligero y sin más azúcar sintáctica que la asignación y la declaración múltiple.
- ***(5 puntos)*** Diseña un lenguaje en nanopass que acepte la representacíon intermedia obtenida del ejercicio anterior.

Notas

- Haz push antes de preguntar por una duda que requiera revisar tu código.
- Para dudas rápidas puedes encontrarme en Telegram.
- Si vas a hacer cambio de integrantes en tu equipo avisame antes.
- Documentación de nanopass.