Compiladores 2023-2
Facultad de Ciencias UNAM
Práctica 4: -> Syntáxis abstracta.

Lourdes del Carmen Gonzáles Huesca Juan Alfonso Garduño Solís
Fausto David Hernández Jasso Juan Pablo Yamamoto Zazueta

13 de marzo
Fecha de Entrega: 22 de marzo

Prelimiares

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. El objetivo de esta práctica es definir 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.

Ejercicios.

- *(10 puntos)* Implementa una función syntax-tree 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**.

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.