Compiladores 2024-2
Facultad de Ciencias, UNAM
Práctica 5: Tabla de símbolos.

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

Fecha de Entrega: 19 de abril

Preliminares.

Para esta práctica vamos a buscar generar la tabla de símbolos para métodos y programas escritos en Jelly. La estructura que vamos a utilizar para la tabla de símbolos será una hash-table como las que usamos en la primer práctica.
En primera instancia parecería fácil, basta con buscar todas las declaraciones de la forma `[var:type]` y agregar a nuestra hash `type` con la llave `var`, sin embargo considera el siguiente programa:

```
1  main{
2      r:int = gdc()
3  }
4  gcd(a:int, b:int):int{
5      r:boolean = false
6      while (r){
7          if (a < b)
8              b = b - a
9          else
10             a = a - b
11         r = a != 0
12     }
13     return b
14 }
```

Claramente es válido, pero si quisieramos generar su tabla de símbolos con la idea mencionada antes habría un inconveniente por la variable `r`: en la sentencia de la linea 2 decimos que `r` tiene tipo int, así agregaríamos a la tabla de símbolos (`r . int`) pero al llegar a la línea 5 agregaríamos (`r` . boolean), en racket cuando agregamos una llave repetida a la tabla se queda el último elemento con el que la vinculamos, así que al terminar el proceso tendríamos en la tabla que `r` es de tipo boolean. Hacer la verificación de tipos con esta información va a causar un problema que en principio no debería existir, por eso antes de la creación de la tabla vamos a renombrar todas las variables de nuestra representación intermedia.
Ten en cuenta que en realidad nuestro código ya no se ve de esta forma pero sirve para ilustrar el posible conflicto, además, hay otras maneras de solucionar este problema pero el renombrado de variables es un recurso real en los procesos de compilación, optimización y asignación de recursos, así que vamos a utilizarlo, de hecho el proceso es similar pero no igual al **single static assignment**.

Implementación:

Para los objetivos de esta práctica primero necesitarás definir un lenguaje en nanopass, este deberá aceptar el syntax-tree resultante de la práctica anterior. Para los siguientes pasos nos vamos a apoyar de dos herramientas que provee nanopass: define-pass y nanopass-case, en el archivo .rkt compartido en google classroom vienen ejemplos de como utilizar ambas herramientas, aún así a continuación se describen las particularidades de estas.

Lenguaje nanopass

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 modifica 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 iniciar la 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

define-pass

Los passes de nanopass son utilizados para definir transformaciones entre los lenguajes definidos, es importante notar que la transformación puede ocurrir dentro de un mismo lenguaje. Para definir los procesos utilizamos la función *define-pass* que está disponible en el dialecto *nanonopass*.
La definición de un proceso, comienza con el identificador de este y una firma. La firma comienza con el lenguaje de entrada y una lista de *formals*, la segunda parte de la firma expecifica el lenguaje de salida junto con el resto de los valores que regresa. Con la siguiente forma:

```
(define-pass pass-name : input-language (formals ...)
                         -> output-languages (extra-return-values ...)
    ...
```

Después del identificador y la firma, se pueden tener mas definiciones, un conjunto de procesadores y el cuerpo del proceso.

```
(define-pass pass-name : input-language (formals ...)
                         -> output-languages (extra-return-values ...)
    definition-clause
    processor ...
    body-expr
)
```

nanopass-case

Como te habrás percatado con define-pass podemos hacer pattern-matching sobre la definición de nuestro lenguaje y modificar la representación intermedia o ejecutar otras acciones a nuestra conveniencia, nanopass-case nos permite hacer lo mismo desde una definición común de racket y de forma modularizada. La sintáxis es la siguiente:

```
(nanopass-case (lang-name variable-del-lenguaje) ir
               matching-clause ...)
```

Dónde matching-clause puede ser de las siguientes formas:

```
[pattern expr ... expr]
[else expr ... expr]
```

Para construir cosas a partir de una representación intermedia (como la tabla de símbolos o una lista con las variables que se encuentran en un programa) te recomiendo usar `nanopass-case`, por otra parte, si quieres modificar la propia representación intermedia utilizar un `define-pass` es lo mejor.

Ejercicios.

- *(5 puntos)* Define un proceso ranme-var que renombre las variables de un programa.
- *(5 puntos)* Define un proceso para generar la tabla de símbolos de un programa symbol-table, este proceso debe aplicarse después de rename-var.