Compiladores 2023-1
Facultad de Ciencias, UNAM
Práctica 5: Tabla de símbolos.

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

12 de abril de 2023
Fecha de Entrega: 28 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 nos vamos a apoyar de dos herramientas que provee nanopass: define-pass y nanopass-case, además de claro, una definición adecuada del lenguaje para el correcto reconocimiento de la representación intermedia actual. En tu repositorio vas a encontrar un archivo `nanopass.rkt`, este contiene los ejemplos generados en la clase del laboratorio con algunos detalles extras, aún así a continuación se describen las particularidades de estas herramientas.

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.