Compiladores 2020-1
Facultad de Ciencias UNAM
Práctica 4: Análisis Sintáctico con Nanopass

Lourdes del Carmen González Huesca Reyes Granados Naomi Itzel
Hernández Luna Nora Hilda

12 de noviembre de 2021
Fecha de entrega: 26 de noviembre de 2021

Lenguaje Fuente

El lenguaje fuente del compilador para esta práctica se define con la siguiente gramática libre del contexto.

```
<programa> ::= <expr>

<expr> ::= <const>
         | <var>
         | (quot <const>)
         | (begin <expr> <expr>*)
         | (primapp <prim> <expr>*)
         | (if <expr> <expr> <expr>)
         | (lambda ([<var> <type>]*) <expr>)
         | (let ([<var> <type> <expr>]*)  <expr>)
         | (letrec ([<var> <type> <expr>]*) <expr>)
         | (list <expr> <expr>*)
         | (<expr> <expr>*)

<const> ::=  <boolean>
         | <integer>
         | <char>

<boolean> ::= #t | #f

<integer> ::= <digit> | <digit><integer>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<var> ::= <car> | <car><var> | <car><digit> | <car><digit><var>

<car> ::= a | b | c | ... | z

<prim> ::= + | - | * | / | length | car | cdr

<type> ::= Bool | Int | Char | List | Lambda
```

Y definido con Nanopass como sigue:

```
(define-language LF
  (terminals
   (variable (x))
   (primitive (pr))
   (constant (c))
   (type (t)))
  (Expr (e body)
    x
    (quot c)
    (begin e* ... e)
    (primapp pr e* ..)
    (if e0 e1 e2)
    (lambda ([x* t*] ...) body)
    (let ([x* t* e*] ...) body)
    (letrec ([x* t* e*] ...) body
    (list e* e)
    (e0 e1 ...)))
```

Que será el lenguaje de entrada para la práctica.

Ejercicios

1. *(2 pts)* Definir el proceso `curry-let` encargado de currificar las expresiones `let` y `letrec`.

   La *currificación* es la técnica encargada de transformar una función de múltiples parámetros en una secuencia de funciones de un único parámetro.

   En este proceso aplicaremos esta técnica a los constructores `let` y `letrec` con el fin de tener una sola asignación en cada uno de estos constructores. Esto nos servirá para la generación de código en C para transformar cada `let` en una asignación y cada `letrec` en una definición de función.

   Para este proceso es necesario definir un nuevo lenguaje `L7` en el que los constructores de `let` y `letrec` tengan una sola asignación.

   ```
   entrada : '(let ([x Int 4] [y Int 6]) (+ x y))
   salida : '(let ([x Int 4]) (let ([y Int 6]) (+ x y)))
   ```

2. *(2 pts)* Definir el proceso `identify-assigments` en el que se detecten los `let` utilizados para definir funciones y se remplazan por `letrec`.

   Este proceso junto con `purify-recursion` de la práctica anterior nos sirven para separar las asignaciones de funciones de las del resto de tipos de dato. A pesar de que no todas las funciones son recursivas, para ésta separación de asignaciones nos será conveniente tener todas las funciones definidas con letrec.

   Para este proceso no es necesario definir ningún lenguaje nuevo, pero si se considera necesario entonces el lenguaje debe tener el nombre `LASG`.

   ```
   entrada : '(let ([foo Lambda (lambda ([x Int]) x)]) (foo 5))
   salida :'(letrec ([foo Lambda (lambda ([x Int]) x)]) (foo 5))
   ```

3. *(2 pts)* Definir el proceso `un-anonymous` encargado de asignarle un identificador a las funciones anónimas (`lambda`).

   Cuando generemos código en C transformaremos todas las `lambda` de nuestro lenguaje en funciones de C, para esto es necesario que todas nuestras funciones tengan un nombre.

   A cada lambda se le asignará un identificador único utilizando un nuevo constructor `letfun` para asignarla.

   Para este proceso es necesario definir un nuevo lenguaje con el nombre `L8` en el cual se agregue el constructor de asignación `letfun` que recibe el identificador de la función, el tipo (que siempre será `Lambda`), una `lambda` que sería el valor y como cuerpo de `letfun` simplemente se llama a la función. De esta forma no se cambia la semántica de la expresión. Hay que notar que no se puede eliminar el constructor `lambda` del lenguaje pues esta sigue siendo una expresión.

```
entrada : '(lambda ([x Bool]) (if x 1 2))
salida : '(letfun ([foo Lambda (lambda ([x Bool]) (if x 1 2))]) foo)
```

Verificadores

4. *(2 pts)* Define el proceso `verify-arity`. Este proceso funciona como verificador de la sintaxis de las expresiones y consiste en verificar que el número de parámetros corresponde con la aridad de las primitivas. Si corresponde, se regresa la misma expresión en caso contrario se lanza un error, especificando que la expresión está mal construida.

   Para este proceso no se debe definir un nuevo lenguaje [1].

```
entrada : '(+ 2 3)
salida : '(+ 2 3)

entrada : '(car 2 3)
salida :  error :  Arity mismatch
```

5. *(2 pts)* Define el proceso `verify-vars`. Este proceso funciona como verificador de la sintaxis de las expresiones y consiste en verificar que la expresión no tenga variables libres, de existir variables libres se regresa un error en caso contrario la salida es la misma expresión.

   Es importante analizar cómo funcionará la recursión en esté proceso, pues que exista una variable libre en una sub-expresión no necesariamente implica que esté libre en toda la expresión.

   Para este proceso no se debe definir un nuevo lenguaje.

   Hint: definir una función auxiliar encargada de encontrar las variables libres de una expresión.

```
entrada : '(+ 2 x)
salida :  error :  Free variable x
```

[1] Hay que observar que la entrada y salida no son tal cual como se muestra en el ejemplo de ejecución, pues las expresiones ya pasaron por una serie de transformaciones en los procesos anteriores del compilado.

Punto extra

La tipificación *à la Church* es un sistema de tipado donde las variables ligadas tienen anotado el tipo de manera explícita. Por ejemplo, la gramática del cálculo lambda con tipos simples *à la Church* sería la siguiente:

$$e \ ::= \ x \mid e \ e \mid \lambda x : T.e$$

donde x pertenece a un conjunto de nombres de variable y T es un tipo.

Algunos ejemplos esperados serían los siguientes:

```
entrada : '(lambda ([x]) (X -> X) x)
salida : '(lambda ([x X]) (X -> X) x)

entrada : '(lambda ([x] [y]) ((X -> Y) -> X -> Y) (x y))
salida : '(lambda ([x X] [y Y]) ((X -> Y) -> X -> Y) (x y))
```

Para obtener un punto extra, escribe en un archivo de texto la descripción de la estrategia que utilizarías para incluir la tipificación à la Church en expresiones generadas por el siguiente lenguaje:

```
(define-language LL
  (terminals
   (variable (x))
   (type (x))
  (Expr (e body)
    x
    (lambda ([x*] ...) t body)
    (e0 e1 ...)))
```

donde los tipos están definidos bajo la siguiente gramática:

$$T \ ::= \ X \mid T \to T$$

Puedes asumir la existencia de los predicados para los terminales y el parser de `LL`.

Con estrategia nos referimos a que indiques si es necesario definir un nuevo lenguaje, cómo serían los procesos del lenguaje `LL` al nuevo lenguaje y qué estrategia deberían utilizar los procesos definidos.

Observaciones

- Los ejercicios deben realizarse en el orden especificado y el lenguaje de entrada de un ejercicio es el lenguaje de salida del ejercicio anterior.
- Pueden definirse todas las funciones auxiliares que se consideren necesarias siempre y cuando se utilicen y estén debidamente comentadas.
- En caso de definir lenguajes nuevos que no hayan sido pedidos de manera explícita, se deberán explicar en los comentarios por qué tomaron esta decisión.
- Pueden importarse todas las bibliotecas que se consideren necesarias, siempre y cuando éstas no resuelvan directamente los problemas.

Entrega

- La entrega será en el *classroom* del curso. Basta entregar los archivos comprimidos y el código bien documentado. Indicando los integrantes del equipo.
- Esta práctica debe realizarse en equipos de a lo más 3 personas.

- En los scripts entregados, deberán considerar incluir lo siguiente como parte de la documentación:
  - Los nombres y números de cuenta de los integrantes del equipo.
  - Descripción detallada del desarrollo de la práctica.
  - En caso de haber sido necesario, la especificación formal de los nuevos lenguajes definidos, utilizando gramáticas.
  - Comentarios, ideas o críticas sobre la práctica.
- Se recibirá la práctica hasta las 23:59:59 horas del día fijado como fecha de entrega, cualquier práctica recibida después no será tomada en cuenta.
- **Cualquier práctica total o parcialmente plagiada, será calificada automáticamente con cero y no se aceptarán más prácticas durante el semestre.**