Compiladores 2021-1
Facultad de Ciencias UNAM
Práctica 3: Preprocesamiento del lenguaje fuente a través de Nanopass

Lourdes del Carmen González Huesca
Juan Alfonso Garduño Solís Nora Hilda Hernández Luna

23 de marzo de 2022
Fechas de entrega: 8 de abril 2022

Introducción

Nanopass es una herramienta que nos permite desarrollar compiladores compuestos de varias fases con un propósito especifico que van traduciendo el código fuente en una serie de lenguajes intermedios bien definidos.

El fin es simplificar y comprender mejor cada paso del compilador y modularlo de tal forma que si se quiere agregar nuevas fases sea sencillo y no implique una reestructuración del proyecto.

A partir de esta práctica se irán desarrollando cada una de estas fases. El objetivo es contar con un compilador completo y funcional al finalizar el curso.

Lenguaje Fuente

Para el desarrollo de las prácticas usaremos como lenguaje fuente minHS. Este último, es un lenguaje de juguete basado en Haskell.
La siguiente gramática libre del contexto define minHS.

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

<expr> ::= <const>
         | <var>
         | <expr> <prim> <expr>
         | begin{<expr>}
         | if(<expr>)then{<expr>}
```

```
          | if(<expr>)then{<expr>}else{<expr>}
          | fun ([<var>:<type>]*:<type>) => <expr>
          | funF (<var> [<var>:<type>]*:<type>) => <
             expr>
          | let ([<var>:<type> = <expr>]*) in  <expr>
             end
          | <expr> app <expr>
          | (<expr>)

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

<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> ::= + | - | * | / | and | or

<type> ::= Bool | Int
```

Implementación

Para continuar con el desarrollo de nuestro compilador, tomamos la implementación del parser que analiza nuestra gramática que describe nuestro lenguaje fuente a algo que *Racket* comprenda y pueda interpretar.

Ahora, procedemos a traducirlo a *Nanopass*.

Lenguajes

La sintaxis para definir lenguajes es la siguiente:

```
(define-language language-name clausula ...)
```

Existen diferentes tipos de cláusulas:

- **Cláusula de extensión** Indica que el lenguaje por definir es una extensión de un lenguaje ya existente. Tiene la siguiente forma:

  ```
  (extends language-name)
  ```

En donde *language-name* es el nombre del lenguaje a extender que debe estar previamente definido. Solo se puede extender de un lenguaje en cada definición.

- **Cláusula Terminal** Es utilizada para determinar las expresiones terminales del lenguaje.

  En un lenguaje que no extiende de otro, la cláusula de terminales tiene es de la forma:

```
(terminals terminal-clause ...)
```

  Donde *terminal-clause* tiene una de las siguientes formas:

```
(terminal-name (meta-var))
```

```
(=> terminal-name (meta-var ...) prettifier)
```

```
(terminal-name (meta-var ...)) => prettifier
```

  - *terminal-name* es el nombre que recibe la expresión terminal. Para que *nanopass* pueda verificar las expresiones terminales es necesario definir, en caso que no exista, el predicado *terminal-name*? que verifica que un elemento de *Racket* sea de este tipo.
  - *meta-var* es el nombre que se le da a la meta variable que se usara como referencia al terminal en el lenguaje y en la definición de las fases.
  - *prettifier* es una expresión de un argumento usada cuando se llama a la función *unparse*[1], para generar una expresión mas legible.

  Cuando la definición del lenguaje extiende a otro, la cláusula **terminals** tiene la siguiente forma:

```
(terminals extended-terminal-clause ...)
```

  donde *extended-terminal-clause* tiene una de las siguientes formas:

```
(+ terminal-name ...)
```

```
(- terminal-name ...)
```

  El símbolo + es para definir una nueva expresión terminal que se añadirá al lenguaje. Y el - indica los terminales que se eliminaran del nuevo lenguaje. Los terminales del lenguaje anterior que no se mencionen en la nueva definición serán copiados tal cual.

[1] Las funciones *parse* y *unparse* se definirán más adelante en el curso

- **Cláusulas no Terminales** Especifica las producciones válidas del lenguaje.

  En lenguajes que no extienden a otros las cláusulas no terminales tienen la siguiente forma:

  ```
  (non-terminal-name (meta-var ...)
  production-clause
  ...)
  ```

  Donde, *non-terminal-name* es un identificador para la producción, *meta-var* es el nombre de una meta-variable utilizada cuando se referencia a las producciones del lenguaje y *production-clause* tiene una de las siguientes formas:

  ```
  terminal-meta-var
  ```

  Una meta-variable terminal que representa por sí sola una producción no terminal.

  ```
  non-terminal-meta-var
  ```

  Una meta-variable no terminal que las formas aceptadas por la producción especifica.

  ```
  production-s-expression
  ```

  Es una *S-expression* que representa un patrón del lenguaje.

En lenguajes que extienden a otros las cláusulas no terminales tienen una forma un poco distinta:

```
(non-terminal-name (meta-var ...)
    extended-production-clause
    ...)
```

En donde *extended-production-form* puede ser

```
(+production-clause ...)
```

```
(-production-clause ...)
```

Los símbolos + y - funcionan de la misma manera que lo hacen en las cláusulas terminales. Si se eliminan todas las producciones de alguna cláusula no terminal en el nuevo lenguaje, entonces se eliminará la cláusula del nuevo lenguaje.

Con lo anterior podemos dar una definición de nuestro lenguaje fuente. La cual sería:

```
(define-language LF
  (terminals
   (variable (x))
   (primitive (pr))
   (constant (c))
   (type (t)))
  (Expr (e body)
    x
    pr
    c
    t
    (begin e* ... e)
    (if e0 e1)
    (if e0 e1 e2)
    (fun ((x* t*) ...) t body* ... body)
    (let ((x* t* e*) ...) body* ... body)
    (funf x ((x* t*) ...) t body* ... body)
    (e0 e1 ...)
    (pr e* ... e)))
```

consiste en un conjunto de expresiones terminales declaradas dentro de la etiqueta terminals:

- variables
- constantes
- primitivas
- tipos

cada una representada con una meta-variable para representarla como una expresión. Y una sola cláusula no terminal la que engloba todas las producciones de la gramática.

Observación: El lenguaje `LF` permite expresiones de distintas aridades para la aplicación y operadores primitivos. Es decir, `LF` es más expresivo que `minHS`.

Para poder verificar las expresiones terminales del lenguaje, *nanopass* requiere un predicado para cada terminal, es decir, una función que verifique si la expresión que recibe es un terminal. Por ejemplo

```
(define (variable? x)
     (symbol? x))
```

Parsers

Con *nanopass* la función encargada de hacer el análisis sintáctico (*parser*) se genera automáticamente con la definición del lenguaje. Solo es necesario definirla como sigue:

```
1 (define-parser parse-LF LF)
```

La función define-parser recibe el identificador del parser y el lenguaje sobre el cual se hará el análisis sintáctico. La función generada recibe una *S-expression* y regresa el árbol de sintaxis abstracta correspondiente.

Al definir la función *parser* se genera también su inversa *unparse* para obtener de nuevo la *S-expression*.

Procesos (Passes)

Los procesos del compilador 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 esta disponible en el dialecto *nanopass*.

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 especifica el lenguaje de salida junto con el resto de los valores de regreso. Con la siguiente forma:

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

Como ejemplo, vamos a definir un proceso que se encargue de hacer implícitamente una expresión *begin* como cuerpo de *let*, *funF* y *fun*.

```
1 (define-pass make-explicit : LF (ir) -> LF ()
2   (Expr : Expr (ir) -> Expr ()
3     [,c `',c]
```

```
4       [(fun ([,x* ,t*] ...) ,t ,[body*] ... ,[body])
5        `(fun ([,x* ,t*] ...) t (begin ,body* ... ,body))
            ]
6       [(let ([,x* ,t* ,[e*]] ...) ,[body*] ... ,[body])
7        `(let ([,x* ,t* ,e*] ...) (begin ,body* ... ,body
            ))]
8       [(funF ,x ([,x* ,t*] ...) ,t ,[body*] ... ,[body])
9        `(funF x ([,x* ,t*] ...) t (begin ,body* ... ,
            body))]))
```

Ejercicios

1. *(0.5 ptos)* Modifica el lexer y parser para que analice la nueva gramática de `minHS`. Observa que el tipo `Func` fue eliminado y se agregaron la expresión `begin` y la condicional `if` de una sola rama.

2. *(1 pto)* Define una función llamada `expr->string` que transforme una expresión e devuelta por nuestro parser en una cadena que represente a e generada por el lenguaje LF. Ejemplos:

```
; Expresión concreta:
; (33 + 2)
> (expr->string (par-exp (prim-exp #<procedure:+> (num-exp 33) (num-exp 2))))
"(+ 33 2)"
; Expresión concreta:
; 3 - (3 / 6)
> (expr->string (prim-exp #<procedure:-> (num-exp 3)
    (par-exp (prim-exp #<procedure:/> (num-exp 3) (num-exp 6)))))
"(- 3 (/ 3 6))"
; Expresión concreta:
; if(#t and #f)then{2}else{3}
> (expr->string (if-exp (prim-exp 'and (bool-exp #t) (bool-exp #f))
    (num-exp 2) (num-exp 3)))
"(if (and #t #f) 2 3)"
```

```
; Expresión concreta:
; fun ([x:Int]:Int) => x
> (expr->string (fun-exp (typeof-exp (brack-exp (typeof-exp (var-exp 'x)
    (int-exp))) (int-exp)) (var-exp 'x)))
"(fun ((x Int)) Int x)"
; Expresión concreta:
; fun ([x:Int][y:Int]:Int) => x*y
> (expr->string (fun-exp
 (typeof-exp (brack-exp (app-t-exp
    (typeof-exp (var-exp 'x)
        (int-exp)) (typeof-exp (var-exp 'y) (int-exp)))) (int-exp))
 (prim-exp #<procedure:*> (var-exp 'x) (var-exp 'y))))
"(fun ((x Int) (y Int)) Int (* x y))"
; Expresión concreta:
; funF (sumita ([x:Int][y:Int]):Int) => x+y
> (fun-f-exp
 (typeof-f-exp (var-exp 'sumita) (brack-exp
    (app-t-exp (typeof-exp (var-exp 'x) (int-exp))
    (typeof-exp (var-exp 'y) (int-exp)))) (int-exp))
 (prim-exp #<procedure:+> (var-exp 'x) (var-exp 'y)))
"(funF sumita ((x Int) (y Int)) Int (+ x y))"
```

3. *(0.5 ptos)* Definir los predicados que falten para que la definición del lenguaje fuente LF funcione correctamente.

4. *(2 pts)* Definir un preproceso del compilador que renombre las variables usadas, de tal forma que queden unificados los nombres de estas y no existan variables repetidas, es decir que no haya variables iguales con diferentes alcances. El nombre de la fase debe ser rename-var (de ser necesario definir un nuevo lenguaje).

   Sugerencia utilizar nombres de variables de la forma $x1, x2, x3, ...$

5. *(3 pts)* Definir un preproceso del compilador que se encargue de eliminar la expresión if sin el caso para *else* unificando las dos producciones que existen para if. El nombre de la fase debe ser remove-one-armed-if (de ser necesario definir un nuevo lenguaje).

6. *(3 pts)* Definir un preproceso del compilador para eliminar las cadenas como elementos terminales del lenguaje. Para esto se deben tratar las cadenas como sinónimos de listas de caracteres como lo hace el lenguaje de programación **Haskell**. El nombre de la fase debe ser remove-string (de ser necesario definir un nuevo lenguaje).

Extras

1. *(Hasta 1.5 pts extra)* Resolver el ejercicio 4 de la sección anterior implementando indices de *De Bruijn*.

Entrega

- La entrega será en el *classroom* del curso. Basta entregar los archivos comprimidos y el código bien documentado. Indicando los integrantes del equipo dentro del readme.pdf (hecho en LaTeX)
- Esta práctica debe realizarse en equipos de a lo más 3 personas.
- En el Readme.pdf (hecho en LaTeX) entregado, 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.**