Universidad Nacional Autónoma de México

Facultad de Ciencias

Práctica 2. Definición y preprocesamiento del lenguaje fuente.

Francisco Arturo Licón Colón
314222095

Kevin Miranda Sánchez
314011163

Victor Hugo Molina Bis
314311212

Compiladores.
Profesor: Lourdes del Carmen González Huesca.
6 de septiembre de 2019

1. Desarrollo del programa

1. **Ejercicio 1.** *Definir los predicados faltantes para que la definicón del lenguaje LF funcione correctamente.*

   Nanopass requiere un predicado para poder verificar las expresiones terminales del lenguaje. El ejemplo que se dio fue el predicado para verificar las variables que están en el conjunto de expresiones terminales del lenguaje LF.

   a) Definición del predicado que verifica las variables.
   ```
   (define (variable? x)
     (symbol? x))
   ```
   La definicón de los demas predicados fue en esencia seguir el ejemplo.

   b) Definición del predicado que verifica las primitivas.
   ```
   (define (primitive? pr)
     (and (memq pr primitives) #t))
   ```
   c) Definición del predicado que verifica las constantes.
   ```
   (define (constant? c)
     (or (integer? c)
         (boolean? c)
         (char? c)))
   ```
   d) Definición del predicado que verifica los tipos.
   ```
   (define (type? t)
     (and (memq t types) #t))
   ```
   Adicionalmnete se tuvieron que definir los primitivos y los tipos del lenguaje.

   a) Se definen los primitivos del lenguaje.
   ```
   (define primitives '(+ - * / and or lenght car cdr))
   ```
   b) Se definen los tipos del lenguaje.
   ```
   (define types '(Bool Int Char List String))
   ```

2. **Ejercicio 3.** *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.*

   Para este ejercicio primero fue necesario definir un nuevo lenguaje, donde eliminamos al if sin el caso del else.

   La definicón de este lenguaje es la siguiente.
   ```
   (define-language L1
     (extends LF)
     (terminals
       (- (primitive (pr)))
       (+ (primitive+void (pr))))
     (Expr (e body)
       (- (if e0 e1))))
   ```
   Este nuevo lenguaje se define con la siguiente gramática.

   $< programa > ::= < expr >$
   $< expr > ::= < const >$
   $\quad | < list >$

$$
\begin{aligned}
&\quad | < var > \\
&\quad | < string > \\
&\quad | (< prim > < const > < const >^*) \\
&\quad | (\text{begin} < expr > < expr >^*) \\
&\quad | (\text{if} < expr > < expr > < expr >) \\
&\quad | (\text{lambda} ([< var > < type >]^*) < expr >) \\
&\quad | (\text{let} ([< var > < type > < expr >]^*) < expr >) \\
&\quad | (\text{letrec} ([< var > < type > < expr >]^*) < expr >) \\
&\quad | (< expr > < expr >^*)
\end{aligned}
$$

$$
\begin{aligned}
< const > ::= &< boolean > \\
&| < integer > \\
&| < char >
\end{aligned}
$$

$< boolean > ::= \#\ \text{t} \mid \#\ \text{f}$

$< integer > ::= < digit > \mid < digit > < integer >$

$< digit > ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$< var > ::= < car > \mid < car > < var > \mid < car > < digit > \mid < car > < digit > < var >$

$< car > ::= \text{a} \mid \text{b} \mid \text{c} \mid ... \mid \text{z}$

$< list > ::= \text{empty} \mid (\text{cons} < const > < list >)$

$< string > ::= \text{""} \mid \text{"} < char > < string > \text{"}$

$< char > ::= \text{a} \mid \text{b} \mid \text{c} \mid ... \mid \text{z} \mid ... \mid @ \mid \# \mid \$ \mid \% \mid \& \mid ...$

$< prim > ::= + \mid - \mid * \mid / \mid \text{and} \mid \text{or} \mid \text{length} \mid \text{car} \mid \text{cdr} \mid \text{void}$

$< type > ::= \text{Bool} \mid \text{Int} \mid \text{Char} \mid \text{List} \mid \text{String}$

Despues se tienen que definir los primitivos agregando void como terminal.

```
(define (primitive+void? pr)
  (and (memq pr primitives+void) #t))
```

Agregamos la definición predicado que verifica los nuevos primitivos.

```
(define (primitive+void? pr)
  (and (memq pr primitives+void) #t))
```

Por último definimos el preproceso.

```
(define-pass remove-one-armed-if : LF (e) -> L1 ()
  (Expr : Expr (e) -> Expr ()
  [(if, [e0], [e1])
  `(if, e0, e1 (void))]))
```

3. **Ejercicio 4.** *Definir un preproceso del compilador para elimiar las cadenas como elementos terminales del lenguaje.*

   Para este ejercicio primero fue necesario definir un nuevo lenguaje, donde eliminamos a las cadenas de los elementos terminales.

   La definicón de este lenguaje es la siguiente.

```
(define-language L2
  (extends L1)
  (terminals
    (- (string (s))))
  (Expr (e body)
    (- s)))
```

   Este nuevo lenguaje se define con la siguiente gramática.

$< programa > ::= < expr >$
$< expr > ::= < const >$
$\quad | < list >$
$\quad | < var >$
$\quad | (< prim > < const > < const >^*)$
$\quad |$ (begin $< expr > < expr >^*$)
$\quad |$ (if $< expr > < expr > < expr >$)
$\quad |$ (lambda ([$< var > < type >$]*) $< expr >$)
$\quad |$ (let ([$< var > < type > < expr >$]*) $< expr >$)
$\quad |$ (letrec ([$< var > < type > < expr >$]*) $< expr >$)
$\quad | (< expr > < expr >^*)$

$< const > ::= < boolean >$
$\quad | < integer >$
$\quad | < char >$

$< boolean > ::=$ # 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

$< list > ::=$ empty | (cons $< const > < list >$)

$< string > ::=$ "" | "$< char > < string >$ "

$< char > ::=$ a | b | c | ... | z | ... | @ | # | $ | % | & | ...

$< prim > ::= + | - | * | / |$ and | or | length | car | cdr | void

$< type > ::=$ Bool | Int | Char | List

Por último definimos el preproceso

```
(define-pass remove-string : L1 (e) -> L2 ()
  (Expr : Expr (e) -> Expr ()
        [,s (string -> list s)]))
```

2. Comentarios

Nanopass resultó ser más complicado de lo esperado, se nos dificultó de manera extrema la realización del ejercicio 2.

La documentación nos ayudo a comprender y así poder desarrollar el ejercicio 1 y conforme se iba jugando con el lenguaje, los ejercicios 3 y 4.

Nanopass a nuestro parecer, no parece ser un lenguaje muy intuitivo de aprender. Y en cuanto a Racket, nuestro mayor problema es su sintaxis, y el uso de muchos parentesís.