Compiladores 2020-1
Facultad de Ciencias UNAM
Práctica 2: Definición y preprocesamiento del lenguaje fuente

Arturo García Campos Alejandro Valderrama Silva

6 de septiembre de 2019

Desarrollo

Para esta práctica primero se definieron los predicados necesarios para poder identificar que tipo de terminal es el que se esá leyendo, por ejemplo si es una variable, una constante, una lista, etc. Como nuestros tipos de *listas* y *strings* serán los mismos que los de racket, no es necesario definirles un predicado, ya que se usan lo definidos en el lenguaje mismo.

Para el desarrollo del 3 y el 4 fue necesario definir nuevos lenguajes, pues la expresión de salida del proceso de precompilación necesita tener otras características que el lenguaje original, LF , no cumple. Más adelante se proporcionan las gramáticas de dichos lenguajes.
Para el 3 lo único que hacemos para eliminar la expresión del if sin el *else* es establecer que en caso que se quiera usar de esta forma regrese una expresión de tipo $if\ then\ else$, en donde *else* será vacío, por lo que se mantiene el significado de la cadena de entrada.
Y para el 4 simplemente se usó la función auxiliar de racket que convierte una string en una lista, de la manera que nosotros buscamos.

Gramática Original

La gramática que describe el lenguaje proporcionado en la práctica es la siguiente:

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

<expr> ::= <const>
         | <list>
         | <var>
         | <string>
         | (<prim> <const> <const>*)
         | (begin <expr> <expr>*)
         | (if <expr> <expr>)
         | (if <expr> <expr> <expr>)
         | (lambda ([<var> <type>]*) <expr>)
```

```
          | (let ([<var> <type> <expr>]*)  <expr>)
          | (letrec ([<var> <type> <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

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

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

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

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

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

Nosotros hicimos modificaciones al lenguaje inicial para quedar finalmente con dos vertientes mÃ¡s del lenguaje, con el objetivo de realizar los problemas 3 y 4.
Es decir, al definir dos nuevos lenguajes (IF y SF) obtuvimos dos nuevas gramÃ¡ticas, las cuales son las siguientes:

- Gramática para el lenguaje IF definido.

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

    <expr> ::= <const>
             | <list>
             | <var>
             | <string>
             | (<prim> <const> <const>*)
             | <void>
             | (begin <expr> <expr>*)
             | (if <expr> <expr> <expr>)
             | (lambda ([<var> <type>]*) <expr>)
             | (let ([<var> <type> <expr>]*)  <expr>)
```

```
          | (letrec ([<var> <type> <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>

<void> ::= v

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

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

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

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

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

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

- Gramática para el lenguaje SF definido.

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

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

    <void> ::= v

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

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

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

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

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

Lenguajes

Entonces los lenguajes que tenemos definidos en nuestro compilador, hasta ahora, son los siguientes

```
(define-language LF
  (terminals
   (variable (x))
   (primitive (pr))
   (constant (c))
   (list (l))
   (string (s))
   (type (t)))
  (Expr (e body)
    x
    pr
    c
    l
    t
    (pr c* ... c)
    (begin e* ... e)
    (if e0 e1)
    (if e0 e1 e2)
```

```
19      (lambda ([x* t*] ...) body* ... body)
20      (let ([x* t* e*] ...) body* ... body)
21      (letrec ([x* t* e*] ...) body* ... body)
22      (e0 e1 ...)))
23
24  (define-language IF
25    (extends LF)
26    (terminals
27      (+ (void (v))))
28    (Expr (e body)
29      (- (if e0 e1))
30      (+ v
31         (void))))
32
33  (define-language SF
34    (extends IF)
35    (terminals
36      (- (string (s))))
37    (Expr (e body)
38      (- s)))
```

Comentarios

Primero nos gustaría aclarar acerca del problema dos, finalmente no pudimos completar el ejercicio por lo que esa sección la dejamos comentada.
En el primer intento lo que intentamos hacer fue definir, dentro de la sección de *definitions*, una función auxiliar para encontrar las variables libres. Pero después de eso no supimos como aplicarlo para avanzar en la sección de ya hacer el renombrado.
Como nos sentimos trabados en el problema nos acercamos a otros equipos para discutir acerca de este ejercicio y coincidimos en algunas cosas con algunos equipos y de ahí logramos definir el segundo intento. Pero tampoco lo queremos poner como entrega porque no es nuestro en realidad.
Queremos aclarar esto para mostrar el esfuerzo del trabajo en intentar este ejercicio y que nuestro objetivo fue intentar de entender como funcionan los cota-morfismos para facilitar las siguientes entregas.

En realidad no tenemos muchos comentarios acerca de la práctica. Nos pareció bastante fácil de entender la manera de definir los lenguajes, junto con su parser, y la manera de probarlos.

Lo que sí nos costó mucho trabajo fue el ejercicio dos. La verdad es que no logramos entender como es que se hace la recursión con del $define - pass$, así como el cómo usar los cota-morfismos para hacer el "pattern matching"de las expresiones a revisar.
Al final no nos salió este ejercicio, pese a que lo intentamos, por lo que decidimos fue mandar el ejercicio comentado para esperar retroalimentación al respecto.