Parseo y Generación de Código – 2do semestre 2018

Licenciatura en Informática con Orientación en Desarrollo de Software

Universidad Nacional de Quilmes

Trabajo práctico 1

Parser para Flecha

Fecha de entrega: 4 de octubre

Índice

1. Introducción

Este TP consiste en implementar un analizador sintáctico para el lenguaje de programación funcional Flecha. El analizador sintáctico puede estar implementado en el lenguaje de su preferencia, y se puede definir manualmente (por ejemplo, usando la técnica de descenso recursivo) o a través un generador de parsers[1]. En el apéndice A se encuentra definida la gramática formal del lenguaje Flecha. Si usan un generador de parsers, probablemente tengan que adaptar la gramática para eliminar ambigüedades.

El analizador sintáctico debe poder analizar programas válidos escritos en lenguaje Flecha y construir un árbol de sintaxis abstracta (AST). Se debe programar también la funcionalidad necesaria para generar el AST en formato `JSON`, como se detalla en el apéndice B, a los efectos de comprobar que su implementación coincida con la esperada.

Nota: En este TP no se evalúan cuestiones de estilo, pero consideren que el TP 2 será una extensión del TP 1, por lo que es recomendable usar buenas prácticas para facilitar su propia tarea en el futuro.

2. Características del lenguaje Flecha

En esta sección describimos informalmente las características del lenguaje Flecha.

- **Tipos.** Flecha es un lenguaje *no tipado*. Los tipos de datos existen únicamente como propiedades de los valores en tiempo de ejecución. Esto es similar a lo que ocurre en lenguajes a veces llamados "dinámicos", como SmallTalk, Python o JavaScript.
- **Tipos de datos primitivos.** Flecha cuenta con dos tipos de datos primitivos: enteros y caracteres. Por ejemplo:

```
1     2     3     0     65536                          -- enteros
'a'   'z'   'A'   'Z'   '0'   '9'   '_'   '\n'         -- caracteres
```

[1] Por ejemplo `yacc` para C/C++, `ANTLR` para Java, `ply` para Python, `happy` para Haskell, etc.

- **Estructuras.** Flecha cuenta con el tipo de las *estructuras*. Una estructura se forma aplicando un constructor a una lista de argumentos. Un constructor es cualquier identificador empezado por mayúsculas. Por ejemplo, las siguientes son estructuras:

```
True                                -- sin argumentos
False                               -- sin argumentos
HeladoDeFrambuesa                   -- sin argumentos
Just 'a'                            -- un argumento
MkCoordenada 3 5                    -- dos argumentos
Cons 1 (Cons 2 (Cons 3 Nil))        -- dos argumentos (recursiva)
```

- **Strings.** Flecha permite escribir listas de caracteres usando una sintaxis abreviada en forma de *string*. Por ejemplo, `"hola"` es una manera abreviada de escribir la siguiente estructura:

```
Cons 'h' (Cons 'o' (Cons 'l' (Cons 'a' Nil)))
```

- **Operaciones aritméticas, relacionales y lógicas.** Flecha soporta los siguientes operadores:

  1. Operaciones **aritméticas** con enteros: suma (`+`), resta (`-`), multiplicación (`*`), división entera (`/`) y resto en la división entera (`%`).
  2. Operaciones **relacionales** para comparar por igualdad (`==`), desigualdad (`!=`), mayor o igual (`>=`), menor o igual (`<=`), mayor estricto (`>`), menor estricto (`<`).
  3. Operaciones **lógicas** para la conjunción, es decir, el "y" lógico (`&&`), la disyunción, es decir, el "o" lógico (`||`), y la negación (`!`).

  Por ejemplo:

```
n * (n - 1) / 2
0 <= x && x <= 9 || 0 <= y && y <= 9
```

- **Definición de constantes y funciones.** Un programa en Flecha es una secuencia de definiciones de constantes y funciones. Por ejemplo:

```
def miNumeroFavorito = 7
def cuadrado x = x * x
def main = cuadrado (cuadrado miNumeroFavorito)
```

- **Alternativa condicional.** Flecha permite escribir expresiones condicionales de la forma `if c then x else y`. Dado que el `if` es una expresión, siempre debe tener una rama `else`. En este TP todavía no nos interesa la *semántica* de las operaciones, pero vale aclarar que la estructura `True` se considera el valor verdadero y la estructura `False` es el valor falso, mientras que cualquier otro valor provocará un error en tiempo de ejecución.

```
if True  then 1 else 2        ==> 1
if False then 1 else 2        ==> 2
if 7     then 1 else 2        ==> (error)
if Nil   then 1 else 2        ==> (error)
```

  La construcción `if` puede incorporar cero, una o varias ramas `elif`:

```
def diaSemana n =
    if n == 1 then Lunes
  elif n == 2 then Martes
  elif n == 3 then Miercoles
  elif n == 4 then Jueves
  elif n == 5 then Viernes
  elif n == 6 then Sabado
              else Domingo
```

- **Pattern matching.** Flecha permite hacer pattern matching usando la construcción `case`. La construcción `case` recibe un valor y tiene varias **ramas**. Cada rama verifica si el valor es una estructura armada usando un constructor. Por ejemplo:

```
def longitud lista =
  case lista
  | Nil       -> 0
  | Cons x xs -> 1 + longitud xs

longitud "abc"    ==> 3
longitud False    ==> (error)
longitud 7        ==> (error)
```

Por simplicidad, supondremos que el *pattern matching* no puede incluir constructores anidados, por ejemplo:

```
-- Lo siguiente no está permitido:
def tieneExactamenteUnElemento lista =
  case lista
  | Nil                -> False
  | Cons x Nil         -> True    -- Nil adentro de Cons
  | Cons x (Cons y ys) -> False   -- Cons adentro de Cons

-- Lo siguiente sí está permitido:
def tieneExactamenteUnElemento lista =
  case lista
  | Nil       -> False
  | Cons x xs -> (case xs
                  | Nil       -> True
                  | Cons y ys -> False)
```

- **Declaraciones locales.** Se permite declarar constantes y funciones locales con `let`.

```
def f x y =
  let z = x + y in
    Coordenada z z

f 10 20  ==> Coordenada 30 30

def g x =
  let h y = x + y in
    Coordenada (g 1) (g 2)

g 10  ==> Coordenada 11 12
```

- **Secuenciación.** A pesar de ser un lenguaje funcional, Flecha cuenta con efectos secundarios para hacer entrada/salida. Para ello resulta útil contar con el operador de secuenciación, escrito con un punto y coma (`;`). Si escribimos `p1; p2` esto tiene el efecto de ejecutar `p1`, descartando su valor, y a continuación ejecutar `p2`. Por ejemplo:

```
def main =
  print "hola\n";
  print "chau\n"
```

El operador de secuenciación es solamente una abreviatura. Internamente el analizador sintáctico convierte la expresión `(p1; p2)` en la expresión `(let _ = p1 in p2)`.

- **Funciones.** En Flecha las funciones también son datos. Se permite también definir funciones anónimas con notación *lambda*. Por ejemplo:

```
def twice f = \ x -> f (f x)

twice (Cons 1) Nil     ==> Cons 1 (Cons 1 Nil)

def map = \ f lista -> case lista
```

```
                        | Nil       -> Nil
                        | Cons x xs -> Cons (f x) (map f xs)

map (\ x -> 2 * x) (Cons 1 (Cons 2 (Cons 3 Nil)))
```

3. Construcción del árbol de sintaxis abstracta

Como resultado del análisis sintáctico, debe construirse un árbol de sintaxis abstracta o AST. La representación interna del AST puede ser la que prefieran, pero deben implementar la funcionalidad necesaria para generar un AST en formato JSON de acuerdo con lo que se especifica en el apéndice B.

4. Pautas de entrega

Para entregar el TP se debe enviar el código fuente por e-mail a la casilla `foones@gmail.com` hasta las 23:59:59 del día estipulado para la entrega, incluyendo `[TP lds-est-parse]` en el asunto y el nombre de los integrantes del grupo en el cuerpo del e-mail. No es necesario hacer un informe sobre el TP, pero se espera que el código sea razonablemente legible. Se debe incluir un README indicando las dependencias y el mecanismo de ejecución recomendado para que el programa provea la funcionalidad pedida. Se recomienda probar el programa con el conjunto de tests provistos.

A. Gramática formal de Flecha

A.1. Convenciones léxicas

En esta sección se detalla el funcionamiento del analizador léxico. Escribimos en ⟨MAYÚSCULAS⟩ el nombre del símbolo terminal o *token* que usaremos formalmente en la gramática.

Blancos y comentarios

Se ignoran los caracteres en blanco, incluyendo espacios (`' '`, caracter `0x20`), tabs (`'\t'`, caracter `0x09`), saltos de línea (`'\n'`, caracter `0x0a`), y retornos de carro (`'\r'`, caracter `0x0d`).

Se ignoran los comentarios. Un comentario puede empezar en cualquier punto del programa con una secuencia de dos guiones seguidos, es decir dos veces el símbolo "menos", (`--`). Un comentario termina en la siguiente ocurrencia de un salto de línea (`'\n'`).

Identificadores

Un **identificador** es una secuencia no vacía de símbolos consecutivos que empiezan con un caracter alfabético y pueden incluir minúsculas (`a..z`), mayúsculas (`A..Z`), caracteres numéricos (`0..9`) y guiones bajos (`_`). Distinguimos dos tipos de identificadores:

1. Identificador que comienza en minúscula, es decir, de la forma `[a-z][_a-zA-Z0-9]*`.
   Se utiliza para nombres de variables, constantes y funciones.
   El token correspondiente es ⟨LOWERID⟩.

2. Identificador que comienza en mayúscula, es decir, de la forma `[A-Z][_a-zA-Z0-9]*`.
   Se utiliza para nombres de constructores.
   El token correspondiente es ⟨UPPERID⟩.

Los identificadores podrían tener longitud arbitrariamente larga, pero se acepta que la implementación se limite a identificadores de longitud hasta 1023.

Constantes numéricas

Una constante numérica es una secuencia no vacía de dígitos decimales (`0..9`). Se utiliza para representar enteros (escritos en decimal). El token correspondiente es ⟨NUMBER⟩.

Las constantes numéricas podrían representar enteros arbitrariamente grandes, pero se acepta que la implementación se limite a números entre 0 y $2^{31} - 1$, es decir, el máximo entero positivo representable usando una representación con signo de 32 bits.

Constantes de caracter

Una constante de caracter consta de un caracter delimitado por comillas simples (`'`). Por ejemplo, `'a'` y `'9'` son constantes de caracter. El token correspondiente es ⟨CHAR⟩. Para poder representar algunos caracteres especiales como retornos de línea, y comillas simples se aceptan las siguientes seis secuencias de escape:

`'\''`	representa la comilla simple `'`
`'\"'`	representa la comilla doble `"`
`'\\'`	representa la contrabarra `\`
`'\t'`	representa un tab `\t`, caracter `0x09`
`'\n'`	representa un salto de línea `\n`, caracter `0x0a`
`'\r'`	representa un retorno de carro `\r`, caracter `0x0d`

Constantes de *string*

Una constante de *string* consta de una secuencia de caracteres delimitados por comillas dobles (`"`). El token correspondiente es ⟨STRING⟩. Para las constantes de *string* se aceptan las mismas seis secuencias de escape que en el caso de las constantes de caracter. Por ejemplo, `"Hola\n"` representa un *string* de cinco caracteres, el último de los cuales es un salto de línea.

Los strings podrían tener longitud arbitrariamente larga, pero se acepta que la implementación se limite a escribir constantes de string de longitud hasta 1023.

Palabras clave

Definiciones

`def`	⟨DEF⟩

Alternativa condicional

`if`	⟨IF⟩
`then`	⟨THEN⟩
`elif`	⟨ELIF⟩
`else`	⟨ELSE⟩

Pattern matching

`case`	⟨CASE⟩

Declaraciones locales

`let`	⟨LET⟩
`in`	⟨IN⟩

Símbolos reservados

Delimitadores

Igual	`=`	⟨DEFEQ⟩	para definiciones (`def x = ...` y `let x = ...`).
Punto y coma	`;`	⟨SEMICOLON⟩	para la secuenciación (`p1; p2`).
Paréntesis izquierdo	`(`	⟨LPAREN⟩	para agrupar expresiones.
Paréntesis derecho	`)`	⟨RPAREN⟩	para agrupar expresiones.
Contrabarra	`\`	⟨LAMBDA⟩	para definir funciones anónimas (`\ x -> x`).
Barra vertical	`|`	⟨PIPE⟩	para las ramas del `case`.
Flecha	`->`	⟨ARROW⟩	para las funciones anónimas y las ramas del `case`.

Operadores lógicos

Conjunción	`&&`	⟨AND⟩
Disyunción	`||`	⟨OR⟩
Negación	`!`	⟨NOT⟩

Operadores relacionales

Igualdad	`==`	⟨EQ⟩
Desigualdad	`!=`	⟨NE⟩
Mayor o igual	`>=`	⟨GE⟩
Menor o igual	`<=`	⟨LE⟩
Mayor estricto	`>`	⟨GT⟩
Menor estricto	`<`	⟨LT⟩

Operadores aritméticos

Suma	`+`	⟨PLUS⟩
Resta	`-`	⟨MINUS⟩
Multiplicación	`*`	⟨TIMES⟩
División	`/`	⟨DIV⟩
Resto	`%`	⟨MOD⟩

A.2. Sintaxis

En esta sección se da una gramática independiente del contexto para Flecha.

⟨*programa*⟩ ⟶ ϵ
| ⟨*programa*⟩ ⟨*definición*⟩

⟨*definición*⟩ ⟶ ⟨DEF⟩ ⟨LOWERID⟩ ⟨*parámetros*⟩ ⟨DEFEQ⟩ ⟨*expresión*⟩

⟨*parámetros*⟩ ⟶ ϵ
| ⟨LOWERID⟩ ⟨*parámetros*⟩

⟨*expresión*⟩ ⟶ ⟨*expresiónExterna*⟩
| ⟨*expresiónExterna*⟩ ⟨SEMICOLON⟩ ⟨*expresión*⟩

⟨*expresiónExterna*⟩ ⟶ ⟨*expresiónIf*⟩
| ⟨*expresiónCase*⟩
| ⟨*expresiónLet*⟩
| ⟨*expresiónLambda*⟩
| ⟨*expresiónInterna*⟩

⟨*expresiónIf*⟩ ⟶ ⟨IF⟩ ⟨*expresionInterna*⟩ ⟨THEN⟩ ⟨*expresionInterna*⟩ ⟨*ramasElse*⟩

⟨*ramasElse*⟩ ⟶ ⟨ELIF⟩ ⟨*expresionInterna*⟩ ⟨THEN⟩ ⟨*expresionInterna*⟩ ⟨*ramasElse*⟩
| ⟨ELSE⟩ ⟨*expresionInterna*⟩

⟨*expresiónCase*⟩ ⟶ ⟨CASE⟩ ⟨*expresionInterna*⟩ ⟨*ramasCase*⟩

⟨*ramasCase*⟩ ⟶ ϵ
| ⟨*ramaCase*⟩ ⟨*ramasCase*⟩

⟨*ramaCase*⟩ ⟶ ⟨PIPE⟩ ⟨UPPERID⟩ ⟨*parámetros*⟩ ⟨ARROW⟩ ⟨*expresionInterna*⟩

⟨*expresiónLet*⟩ ⟶ ⟨LET⟩ ⟨ID⟩ ⟨*parámetros*⟩ ⟨DEFEQ⟩ ⟨*expresiónInterna*⟩ ⟨IN⟩ ⟨*expresiónExterna*⟩

⟨*expresiónLambda*⟩ ⟶ ⟨LAMBDA⟩ ⟨*parámetros*⟩ ⟨ARROW⟩ ⟨*expresiónExterna*⟩

⟨*expresiónInterna*⟩ ⟶ ⟨*expresiónAplicación*⟩
| ⟨*expresiónInterna*⟩ ⟨*operadorBinario*⟩ ⟨*expresiónInterna*⟩
| ⟨*operadorUnario*⟩ ⟨*expresiónInterna*⟩

⟨*operadorBinario*⟩ ⟶ ⟨AND⟩
| ⟨OR⟩
| ⟨EQ⟩
| ⟨NE⟩
| ⟨GE⟩
| ⟨LE⟩
| ⟨GT⟩
| ⟨LT⟩
| ⟨PLUS⟩
| ⟨MINUS⟩
| ⟨TIMES⟩
| ⟨DIV⟩
| ⟨MOD⟩

⟨*operadorUnario*⟩ ⟶ ⟨NOT⟩
| ⟨MINUS⟩

⟨*expresiónAplicación*⟩ ⟶ ⟨*expresiónAtómica*⟩
| ⟨*expresiónAplicación*⟩ ⟨*expresiónAtómica*⟩

⟨*expresiónAtómica*⟩ ⟶ ⟨LOWERID⟩
| ⟨UPPERID⟩
| ⟨NUMBER⟩
| ⟨CHAR⟩
| ⟨STRING⟩
| ⟨LPAREN⟩ ⟨*expresión*⟩ ⟨RPAREN⟩

Aclaraciones:

- El operador de secuenciación (punto y coma) tiene menor precedencia que todas las demás construcciones. Por ejemplo, son equivalentes:

```
if a then b else c ; d
(if a then b else c) ; d
```

- Le siguen en precedencia las *expresiones externas*, es decir, el `if`, el `case`, el `let` y las funciones anónimas (lambda). Por ejemplo, son equivalentes:

```
if a then b else c + d
if a then b else (c + d)
```

- La condición y las ramas del `if` y del `case` no pueden ser expresiones externas, salvo que estén entre paréntesis. Asimismo, el valor ligado en un `let` no puede ser una expresión externa. Por ejemplo,

```
if x then \ x -> x   else y        -- error de sintaxis
if x then (\ x -> x) else y        -- OK

case dir
| Izq -> if a then b else c      -- error de sintaxis

case dir
| Izq -> (if a then b else c)    -- OK

let x = let y = 3 in y   in x      -- error de sintaxis
let x = (let y = 3 in y) in x      -- OK
```

- El cuerpo del `let` y el cuerpo de las funciones anónimas (lambda) pueden ser expresiones externas. Por ejemplo:

```
let x = 1 in
let y = 2 in
  \ z -> if a then x else y + z    -- OK
```

- Todos los operadores tienen menor precedencia que la aplicación. Por ejemplo, son equivalentes:

```
f x y + g z
(f x y) + (g z)
```

A.3. Asociatividad y precedencia

Los operadores binarios y unarios deben respetar la siguiente tabla, ordenada de menor a mayor precedencia. Los elementos de cada fila tienen la misma precedencia entre ellos, y mayor precedencia que los de las filas anteriores. Todos los operadores binarios son asociativos a izquierda.

`||`					
`&&`					
`!` *(unario)*					
`==`	`!=`	`>=`	`<=`	`>`	`<`
`+`	`-`				
`*`					
`/`	`%`				
`-` *(unario)*					

Por ejemplo, son equivalentes:

```
! x == y && z > a * b + c * d
(!(x == y)) && (z > ((a * b) + (c * d)))
```

B. Árbol de sintaxis abstracta

El árbol de sintaxis abstracta en formato JSON tiene la siguiente estructura. Usamos `ID` para denotar un identificador arbitrario (string) y `NUM` para denotar un número arbitrario (entero).

`Program`	::=	`[ Definition, ..., Definition ]`	Lista de n definiciones, con $n \geq 0$.
`Definition`	::=	`["Def", ID, Expr]`	Definición.
`Expr`	::=	`["ExprVar", ID]`	Variable.
	|	`["ExprConstructor", ID]`	Constructor.
	|	`["ExprNumber", NUM]`	Constante numérica.
	|	`["ExprChar", NUM]`	Constante de caracter.
	|	`["ExprCase", Expr, [CaseBranch, ..., CaseBranch]]`	Case de n ramas, con $n \geq 0$.
	|	`["ExprLet", ID, Expr, Expr]`	Declaración local.
	|	`["ExprLambda", ID, Expr]`	Función anónima.
	|	`["ExprApply", Expr, Expr]`	Aplicación.

`CaseBranch` ::= `["CaseBranch", ID, [ID, ..., ID], Expr]` Rama del case de n parámetros, con $n \geq 0$.

- Todas las lambdas reciben un único parámetro. Las lambdas que reciben varios parámetros deben expresarse como muchas lambdas anidadas. Por ejemplo, son equivalentes:

```
\ x y z -> x + y
\ x -> (\ y -> (\ z -> x + y))
```

- Las funciones definidas con `def` y `let` se deben expresar con lambdas. Por ejemplo, son equivalentes:

```
def f x y = let g z = x * z in y + z
def f = \ x y -> let g = (\ z -> x * z) in y + z
```

- El operador de secuenciación debe expresarse en términos del `let`. Por ejemplo, son equivalentes:

```
def main = print "hola\n"; print "chau\n"
def main = let _ = print "hola\n" in print "chau\n"
```

- Las ramas `elif` de un `if` deben expresarse internamente como ifs anidados. Por ejemplo, son equivalentes:

```
if a then 1 elif b then 2 elif c then 3 else 4
if a then 1 else (if b then 2 else (if c then 3 else 4))
```

- El `if` debe expresarse internamente como un `case`. Por ejemplo, son equivalentes:

```
if a then b else c

case a
| True  -> b
| False -> c
```

- Los caracteres se expresan con su código en la codificación ASCII o UTF-8. Se puede asumir que la entrada cuenta exclusivamente con caracteres imprimibles (en el rango 32..127). Por ejemplo, el AST de la expresión `'A'` es `["ExprChar", 97]`.

- Los strings se expresan como listas de caracteres construidas usando `Cons` y `Nil`. Por ejemplo, son equivalentes:

```
"hola"
Cons 'h' (Cons 'o' (Cons 'l' (Cons 'a' Nil)))
```

- La aplicación es siempre binaria, usando currificación. Por ejemplo, son equivalentes:

```
f (g 1) 2 (g 3)
((f (g 1)) 2) (g 3)
```

- Los operadores se expresan como la aplicación de *variables* con nombres especiales a sus argumentos. Por ejemplo:

  - El AST de la expresión `!4` es:

    `["ExprApply", ["ExprVar", "NOT"], ["ExprNumber", 4]]`

  - El AST de la expresión `x + y` es:

    `["ExprApply", ["ExprApply", ["ExprVar", "ADD"], ["ExprVar", "x"]], ["ExprVar", "y"]]`

  Los nombres de los demás operadores son los siguientes:

`||`	`"OR"`	`&&`	`"AND"`	`!` *(unario)*	`"NOT"`
`==`	`"EQ"`	`!=`	`"NE"`	`>=`	`"GE"`
`<=`	`"LE"`	`>`	`"GT"`	`<`	`"LT"`
`+`	`"ADD"`	`-`	`"SUB"`	`*`	`"MUL"`
`/`	`"DIV"`	`%`	`"MOD"`	`-` *(unario)*	`"UMINUS"`

  Observar que no puede haber conflicto con los nombres de otras variables, porque las variables siempre tienen nombres empezados en minúsculas.

B.1. Ejemplo completo de análisis sintáctico

Programa

```
def sumar lista =
  case lista
  | Nil       -> 0
  | Cons x xs -> x + sumar xs

def main = sumar (Cons 1 (Cons 2 Nil))
```

AST obtenido

```
[
 ["Def", "sumar",
  ["ExprLambda", "lista",
   ["ExprCase",
    ["ExprVar", "lista"],
    [
     ["CaseBranch", "Nil", [],
      ["ExprNumber", 0]
     ],
     ["CaseBranch", "Cons", ["x", "xs"],
      ["ExprApply",
       ["ExprApply",
        ["ExprVar", "ADD"],
        ["ExprVar", "x"]
       ],
       ["ExprApply",
        ["ExprVar", "sumar"],
        ["ExprVar", "xs"]
       ]
      ]
     ]
    ]
   ]
  ]
 ],
 ["Def", "main",
  ["ExprApply",
   ["ExprVar", "sumar"],
   ["ExprApply",
    ["ExprApply",
     ["ExprConstructor", "Cons"],
     ["ExprNumber", 1]
    ],
    ["ExprApply",
     ["ExprApply",
      ["ExprConstructor", "Cons"],
      ["ExprNumber", 2]
     ],
     ["ExprConstructor", "Nil"]
    ]
   ]
  ]
 ]
]
```