Parseo y Generación de Código – 2do semestre 2019

Licenciatura en Informática con Orientación en Desarrollo de Software

Universidad Nacional de Quilmes

Trabajo práctico 1

Parser para Avalancha

Fecha de entrega: jueves 3 de octubre

Índice

1. Introducción

Este TP consiste en implementar un analizador sintáctico para el lenguaje Avalancha. 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 de Avalancha. 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 Avalancha 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. Descripción informal

2.1. Avalancha como lenguaje de programación

Avalancha es un lenguaje puramente funcional[2] cuyo único tipo de datos son los **árboles**. Los árboles se construyen usando constructores (cuyos nombres comienzan siempre en mayúsculas) aplicados a argumentos, que a su vez son árboles. Un constructor puede recibir cualquier número de argumentos (0, 1, o varios). Por ejemplo, los siguientes son ejemplos de árboles conformados aplicando constructores a parámetros:

`Nil`	el constructor `Nil` aplicado a 0 argumentos
`Nil()`	otra manera de escribir el constructor `Nil` aplicado a 0 argumentos
`Bin(Nil, A, Nil)`	el constructor `Bin` aplicado a tres argumentos
`Cons(Suc(Suc(Zero)), Nil)`	etc.

[1] Por ejemplo `yacc` para C/C++, `ANTLR` para Java, `ply` para Python, `happy` para Haskell, etc.
[2] Emparentado con la noción de *Recursive Program Scheme*.

En Avalancha se pueden definir funciones recursivas para procesar árboles utilizando *pattern matching*. Por ejemplo, el siguiente programa define una función `length` que computa la longitud de una lista (construida con `Nil` y `Cons(-,-)`) y devuelve un número natural (construido con `Zero` y `Suc(-)`), y a continuación verifica que la longitud de la lista `[A, B, C]` sea 3:

```
fun length
  Nil         -> Zero
  Cons(_, xs) -> Suc(length(xs))

check length(Cons(A, Cons(B, Cons(C, Nil)))) == Suc(Suc(Suc(Zero)))
```

Más precisamente, un programa siempre consta de una secuencia de **declaraciones** de funciones, seguida de una secuencia de **chequeos**.

La **declaración** de una función siempre es de la forma

```
fun nombre
  regla1
  ...
  reglaN
```

Una función puede tener una cantidad arbitraria de reglas (0, 1, o varias). Cada regla es de la forma:

```
patrón1, ..., patrónN -> expresión
```

Cada regla puede involucrar una cantidad arbitraria de patrones (0, 1, o varios).

En Avalancha es posible definir funciones que reciben varios parámetros, y pueden hacer pattern matching contra varios de ellos. Por ejemplo la función `suma` recibe dos números naturales y calcula la suma haciendo recursión en el primer parámetro; la función `zipWithSuma` recibe dos listas de la misma longitud y devuelve la suma de sus elementos, componente a componente:

```
fun suma
  Zero  , y -> y
  Suc(x), y -> Suc(suma(x, y))

fun zipWithSuma
  Nil        , Nil         -> Nil
  Cons(x, xs), Cons(y, ys) -> Cons(suma(x, y), zipWithSuma(xs, ys))
```

Notar que no hay funciones "de alto orden", es decir, las variables `x`, `xs`, etc. siempre representan árboles, y a las funciones `suma`, `zipWithSuma`, etc. sólo se las puede utilizar por su nombre al invocarlas.

El lenguaje Avalancha es no tipado. Por ejemplo, nada impide escribir el árbol `Cons(Zero, Zero)`. Además, los patrones en una declaración de función podrían no cubrir todos los casos. Por ejemplo la siguiente función `second` recibe una lista y devuelve su segundo elemento, y no contempla otros casos (ej. la lista vacía `Nil`):

```
fun second
  Cons(_, Cons(x, _)) -> x
```

Además nada impide que haya patrones superpuestos o patrones redundantes. Por ejemplo, en la definición de `y` los primeros dos patrones se superponen, y en la definición de `head` el segundo patrón es redundante:

```
fun y
  False, x      -> x
  x    , False  -> x
  True , True   -> True

fun head
  Cons(x, xs)  -> x
  Cons(x, Nil) -> x
```

2.2. Avalancha como lenguaje de verificación

El objetivo de Avalancha es **avalar** que un programa cumple con alguna especificación formal. Para eso es posible acompañar a todas (o algunas) de las funciones con su pre y su postcondición, utilizando fórmulas lógicas. Todas las funciones pueden incluir antes de las reglas los siguientes tres elementos, en el siguiente orden (todos ellos son opcionales):

1. La **signatura** que se indica con dos puntos ("`:`") y sirve para darle nombre a los parámetros y al resultado.
2. La **precondición** que se indica con un signo de interrogación ("`?`") y se acompaña de una fórmula lógica.
3. La **postcondición** que se indica con un signo de exclamación ("`!`") y se acompaña de una fórmula lógica.

Por ejemplo, el siguiente programa:

```
fun sonIguales
  : x , y -> z
  ? ((x == A) or (x == B)) and ((y == A) or (y == B))
  ! ((x == y) and (z == True)) or (not (x == y) and (z == False))
  A , A -> True
  A , B -> False
  B , A -> False
  B , B -> True
```

tiene:

1. Una signatura "`: ...`" que indica que la función recibe dos parámetros (llamados `x` e `y`) y devuelve un resultado (llamado `z`).
2. Una precondición "`? ...`" que requiere que los dos parámetros de la función sean elementos del conjunto $\{\texttt{A}, \texttt{B}\}$.
3. Una postcondición "`! ...`" que asegura que el resultado de la función es `True` si y sólo si sus dos parámetros son iguales, y `False` en caso contrario.

2.3. 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.

3. 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

A.1. Convenciones léxicas

En esta sección se detalla la sintaxis léxica de Avalancha, es decir cómo se escriben los lexemas o *tokens*. 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.

Palabras clave

Declaraciones

`fun`	⟨FUN⟩	para declarar funciones.
`check`	⟨CHECK⟩	para escribir chequeos.

Operadores lógicos

`imp`	⟨IMP⟩	implicación "**`a imp b`**".
`and`	⟨AND⟩	conjunción "**`a and b`**".
`or`	⟨OR⟩	disyunción "**`a or b`**".
`not`	⟨NOT⟩	negación "**`not a`**".
`true`	⟨TRUE⟩	verdadero "**`true`**"
`false`	⟨FALSE⟩	falso "**`false`**"

Símbolos reservados

Delimitadores

Paréntesis izquierdo	`(`	⟨LPAREN⟩	para agrupar fórmulas y expresiones.
Paréntesis derecho	`)`	⟨RPAREN⟩	para agrupar fórmulas y expresiones.
Coma	`,`	⟨COMMA⟩	para separar parámetros.
Flecha	`->`	⟨ARROW⟩	para separar los patrones del cuerpo de una regla.

Expresiones

Comodín	`_`	⟨UNDERSCORE⟩	para los patrones comodín.

Lenguaje de verificación

Dos puntos	`:`	⟨COLON⟩	para marcar la signatura.
Signo de interrogación	`?`	⟨QUESTION⟩	para marcar la precondición.
Signo de exclamación	`!`	⟨BANG⟩	para marcar la postcondición.

Fórmulas

Igualdad	`==`	⟨EQ⟩	para comparar expresiones en una fórmula.

A.2. Sintaxis

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

⟨*programa*⟩ ⟶ ⟨*declaraciones*⟩ ⟨*chequeos*⟩

⟨*declaraciones*⟩ ⟶ ϵ
| ⟨*declaración*⟩ ⟨*declaraciones*⟩

⟨*chequeos*⟩ ⟶ ϵ
| ⟨*chequeo*⟩ ⟨*chequeos*⟩

⟨*declaración*⟩ ⟶ ⟨FUN⟩ ⟨LOWERID⟩ ⟨*signatura*⟩ ⟨*precondición*⟩ ⟨*poscondición*⟩ ⟨*reglas*⟩

⟨*chequeo*⟩ ⟶ ⟨CHECK⟩ ⟨*fórmula*⟩

⟨*signatura*⟩ ⟶ ϵ — *la signatura es opcional*
| ⟨COLON⟩ ⟨*listaParámetros*⟩ ⟨ARROW⟩ ⟨*parámetro*⟩

⟨*precondición*⟩ ⟶ ϵ — *la precondición es opcional*
| ⟨QUESTION⟩ ⟨*fórmula*⟩

⟨*poscondición*⟩ ⟶ ϵ — *la postcondición es opcional*
| ⟨BANG⟩ ⟨*fórmula*⟩

⟨*listaParámetros*⟩ ⟶ ϵ — *la signatura puede no tener parámetros, ej.* `fun f : -> resultado`
| ⟨*listaParámetrosNoVacía*⟩

⟨*listaParámetrosNoVacía*⟩ ⟶ ⟨*parámetro*⟩
| ⟨*parámetro*⟩ ⟨COMMA⟩ ⟨*listaParámetrosNoVacía*⟩

⟨*parámetro*⟩ ⟶ ⟨UNDERSCORE⟩ — *el parámetro puede ser un comodín*
| ⟨LOWERID⟩ — *nombre del parámetro (en minúscula)*

⟨*reglas*⟩ ⟶ ϵ — *una función puede no tener ninguna regla*
| ⟨*regla*⟩ ⟨*reglas*⟩

⟨*regla*⟩ ⟶ ⟨*listaPatrones*⟩ ⟨ARROW⟩ ⟨*expresión*⟩

⟨*patrón*⟩ ⟶ ⟨UNDERSCORE⟩ — *comodín*
| ⟨LOWERID⟩ — *variable, ej.* `x`
| ⟨UPPERID⟩ — *constructor solo, ej.* `Zero`
| ⟨UPPERID⟩ ⟨LPAREN⟩ ⟨*listaPatrones*⟩ ⟨RPAREN⟩ — *constructor con parámetros, ej.* `Cons(x, xs)`

⟨*listaPatrones*⟩ ⟶ ϵ — *puede ser vacía, ej.* `fun f -> A`
| ⟨*listaPatronesNoVacía*⟩

⟨*listaPatronesNoVacía*⟩ ⟶ ⟨*patrón*⟩
| ⟨*patrón*⟩ ⟨COMMA⟩ ⟨*listaPatronesNoVacía*⟩

⟨*expresión*⟩ ⟶ ⟨LOWERID⟩ — *variable, ej.* `x`
| ⟨LOWERID⟩ ⟨LPAREN⟩ ⟨*listaExpresiones*⟩ ⟨RPAREN⟩ — *invocación a función, ej.* `f(x, True)`
| ⟨UPPERID⟩ — *constructor solo, ej.* `Zero`
| ⟨UPPERID⟩ ⟨LPAREN⟩ ⟨*listaExpresiones*⟩ ⟨RPAREN⟩ — *constructor con parámetros, ej.* `Cons(x, xs)`

⟨*listaExpresiones*⟩ ⟶ ϵ — *puede ser vacía, ej.* `f()`
| ⟨*listaExpresionesNoVacía*⟩

⟨*listaExpresionesNoVacía*⟩ ⟶ ⟨*expresión*⟩
| ⟨*expresión*⟩ ⟨COMMA⟩ ⟨*listaExpresionesNoVacía*⟩

⟨*fórmula*⟩ ⟶ ⟨*fórmulaImpOrAndNeg*⟩

⟨*fórmulaImpOrAndNeg*⟩ ⟶ ⟨*fórmulaOrAndNeg*⟩
| ⟨*fórmulaOrAndNeg*⟩ ⟨IMP⟩ ⟨*fórmulaImpOrAndNeg*⟩

⟨*fórmulaOrAndNeg*⟩ ⟶ ⟨*fórmulaAndNeg*⟩
| ⟨*fórmulaAndNeg*⟩ ⟨OR⟩ ⟨*fórmulaOrAndNeg*⟩

⟨*fórmulaAndNeg*⟩ ⟶ ⟨*fórmulaNeg*⟩
| ⟨*fórmulaNeg*⟩ ⟨AND⟩ ⟨*fórmulaAndNeg*⟩

⟨*fórmulaNeg*⟩ ⟶ ⟨*fórmulaAtómica*⟩
| ⟨NOT⟩ ⟨*fórmulaNeg*⟩

⟨*fórmulaAtómica*⟩ ⟶ ⟨TRUE⟩
| ⟨FALSE⟩
| ⟨LPAREN⟩ ⟨*fórmula*⟩ ⟨RPAREN⟩
| ⟨*expresion*⟩ — *NOTA: una expresión sola* `e` *equivale a la fórmula* `e == True`.
| ⟨*expresion*⟩ ⟨EQ⟩ ⟨*expresion*⟩

NOTA: si una fórmula consta de una expresión sola `e`, el parser debe generar un AST como si la fórmula fuera de la forma `e == True`. Por ejemplo, las dos líneas siguientes deberían generar el mismo AST:

```
check f()
check f() == True
```

Ver el resultado del test `test05.input` en `test05.expected`.

A.3. Asociatividad y precedencia

La implicación, la disyunción y la implicación son asociativas a derecha. Por ejemplo: "**`a imp b imp c imp d`**" es equivalente a "**`(a imp (b imp (c imp d)))`**".

Los operadores lógicos, de menor a mayor precedencia, son:

1. Implicación (`imp`).
2. Disyunción (`or`).
3. Conjunción (`and`).
4. Negación (`not`).

Por ejemplo, "**`a or not b and c imp d`**" es equivalente a "**`(a or ((not b) and c)) imp d`**".

A.4. Restricciones sobre la gramática

Una vez construido el árbol de sintaxis abstracta, el programa debería chequear que se cumplan las siguientes condiciones sobre el programa. En caso de que no se cumplan, hay un error de sintaxis.

- [RO1] — Todas las reglas de una función deben tener la misma cantidad de parámetros. Por ejemplo, el siguiente programa se rechaza porque de acuerdo con la primera regla la función recibe 2 parámetros, pero de acuerdo con la segunda regla la función recibe 1 parámetro:

```
fun f
  A , B -> C
  D     -> E
```

En caso de que la función tenga signatura, la aridad de la signatura también debe coincidir con la aridad de todas las reglas. Por ejemplo, el siguiente programa se rechaza porque de acuerdo con la signatura recibe 2 parámetros, pero de acuerdo con las reglas recibe 1 parámetro:

```
fun f : x , y -> z
  A -> B
```

El siguiente programa se acepta:

```
fun f             -- Función sin signatura ni reglas
fun g : x -> y    -- Función con signatura y sin reglas
fun h             -- Función sin signatura y con reglas.
  A -> B
fun i : x -> y    -- Función con signatura y con reglas que coinciden en su aridad.
  A -> B
```

- [R02] — No pueden existir dos funciones con el mismo nombre. Por ejemplo, el siguiente programa se rechaza porque hay dos definiciones para la función `f`:

  ```
  fun f A -> B
  fun f C -> D
  ```

  Notar que una variable podría llamarse igual que una función. Por ejemplo, el siguiente programa se acepta. Notar que `foo` se usa para el nombre de una función y también para el nombre de un parámetro. En `foo(foo)` el primero es una función y el segundo es un parámetro:

  ```
  fun foo
    A -> B

  fun head
    Cons(foo, _) -> foo(foo)
  ```

- [R03] — Los patrones de una misma regla no pueden incluir variables repetidas. Por ejemplo, el siguiente programa se rechaza porque la `x` aparece repetida en el patrón de una misma regla:

  ```
  fun f
    A(x, x) -> B
  ```

  El siguiente programa también se rechaza por el mismo motivo:

  ```
  fun f
    A(x, y), B(x) -> C
  ```

  Notar que el siguiente programa se acepta porque la `x` aparece repetida pero nunca aparece dos veces en los patrones de la misma regla:

  ```
  fun f
    A(x) -> x
    B(x) -> C(x, x, x)
  ```

  Notar también que se pueden usar varios patrones comodín en la misma regla. El siguiente programa se acepta:

  ```
  fun f
    A(_, _) -> B
  ```

- [R04] — No puede haber variables repetidas en la signatura de una función. Por ejemplo, el siguiente programa se rechaza porque la variable `y` aparece dos veces en la signatura:

  ```
  fun f : x , y -> y
  ```

B. Formato del AST en JSON

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

`Program` ::= `["program", Declarations, Checks]` — *Lista de tamaño 2, con declaraciones y chequeos.*

`Declarations` ::= `[Declaration, ..., Declaration]` — *Lista con n declaraciones, para $n \geq 0$.*

`Checks` ::= `[Check, ..., Check]` — *Lista con n chequeos, para $n \geq 0$.*

`Declaration` ::= `["fun", ID, Signature, Precondition, Postcondition, Rules]`

`Signature` ::= `["sig", [ID, ..., ID], ID]` — *Nombres de los n parámetros ($n \geq 0$) seguidos del nombre del resultado. Cualquiera de los nombres puede ser `""` si es un comodín. Si la función no tiene signatura explícita, se debe completar con comodines con la aridad correspondiente.*

`Precondition` ::= `["pre", Formula]` Si la función no tiene precondición, se usa la fórmula `["true"]`.

`Postcondition` ::= `["post", Formula]` Si la función no tiene precondición, se usa la fórmula `["true"]`.

`Rules` ::= `[Rule, ..., Rule]` — *Lista con n reglas, para $n \geq 0$.*

`Check` ::= `["check", Formula]`

`Rule` ::= `["rule", Patterns, Expr]`

`Patterns` ::= `[Pattern, ..., Pattern]` — *Lista con n patrones, para $n \geq 0$.*

`Formula` ::= `["imp", Formula, Formula]` — *Implicación.*
| `["or", Formula, Formula]` — *Disyunción.*
| `["and", Formula, Formula]` — *Conjunción.*
| `["not", Formula, Formula]` — *Negación.*
| `["equal", Expr, Expr]` — *Comparación por igualdad.*
| `["true"]` — *Verdadero.*
| `["false"]` — *Falso.*

`Expr` ::= `["var", ID]` — *Variable.*
| `["cons", ID, [Expr, ..., Expr]]` — *Constructor con n argumentos ($n \geq 0$).*
| `["app", ID, [Expr, ..., Expr]]` — *Función con n argumentos ($n \geq 0$).*

`Pattern` ::= `["pwild"]` — *Comodín.*
| `["pvar", ID]` — *Variable.*
| `["pcons", ID, [Expr, ..., Expr]]` — *Constructor con n argumentos ($n \geq 0$).*