Parseo y Generación de Código – 2do semestre 2017

Licenciatura en Informática con Orientación en Desarrollo de Software

Universidad Nacional de Quilmes

Trabajo práctico 1

El parser genérico Lleca

Fecha de entrega: 12 de octubre

Índice

1. Introducción

Este TP consiste en implementar un lenguaje genérico de análisis sintáctico que llamaremos Lleca. El parser Lleca recibe como entrada una gramática que describe un lenguaje. Por ejemplo, el archivo **robot.ll** contiene la especificación de un lenguaje para controlar un robot:

robot.ll

```
programa
| /* programa vacío */  => Fin
| comando programa      => Secuencia($1, $2)

comando
| "AVANZAR" NUM         => CmdAvanzar($2)
| "GIRAR" sentido       => CmdGirar($2)

sentido
| "IZQ"                 => Izquierda
| "DER"                 => Derecha
```

Cada producción de la gramática está acompañada de una acción. Las acciones se escriben después de una flecha "=>" e indican cuál es el resultado que se debe asociar a dicha construcción gramatical. El resultado asociado a cada construcción es siempre un árbol. El parser Lleca recibe además como entrada un archivo fuente, por ejemplo el siguiente:

esquina.input

```
AVANZAR 10 GIRAR DER AVANZAR 10
```

El parser Lleca analiza el archivo fuente con la gramática indicada, y produce un árbol como resultado. Por ejemplo, si se invoca al parser con la gramática **robot.ll** y el archivo fuente **esquina.input**, se obtiene como resultado el siguiente árbol:

```
Secuencia( CmdAvanzar(10),
           Secuencia( CmdGirar(Derecha),
                      Secuencia( CmdAvanzar(10),
                                 Fin ) ) )
```

En caso de que el archivo fuente no esté en el lenguaje generado por la gramática, Lleca debe emitir un mensaje de error. En el contexto de este TP no es necesario que los mensajes de error sean descriptivos[1]; alcanzará con indicar que hay un error de sintaxis.

2. Análisis léxico

En este apartado se describe cómo debe funcionar el analizador léxico o tokenizador. El mismo analizador léxico se utilizará para segmentar tanto la gramática como el archivo fuente. Los archivos que manipula Lleca constan de una secuencia de tokens:

- **Identificadores**. Son simplemente nombres. En distintos lenguajes y contextos pueden representar diferentes cosas; por ejemplo, nombres de variables o funciones. Los identificadores son de la forma `[a-zA-Z_][a-zA-Z0-9_]*` es decir, constan de:
  - Un primer caracter que debe ser alfabético (`a, ..., z, A, ..., Z`) o un guión bajo (`_`).
  - Seguido de una secuencia de 0 o más caracteres que pueden ser alfabéticos (`a, ..., z, A, ..., Z`), dígitos (`0, ..., 9`) o un guión bajo (`_`).

  Ejemplos de identificadores:

  `x` `y` `foo` `BAR` `_` `CamelCase` `f_o_o` `foo42`

  Idealmente los identificadores deberían ser de longitud ilimitada, pero se acepta que la implementación se restrinja a identificadores de longitud entre 1 y 63.

- **Números.** Son constantes numéricas (enteros no negativos). Son de la forma `[0-9]+` es decir, constan de una secuencia de uno o más dígitos.

  Ejemplos de constantes numéricas:

  `0` `1` `2` `001` `42` `123456789`

  Idealmente las constantes numéricas deberían ser de longitud ilimitada, pero se acepta que la implementación se restrinja a constantes numéricas entre 0 y $2^{31} - 1$, que corresponde al máximo entero positivo representable en un entero de 32 bits con signo.

[1] Pero cuanto más descriptivos, mejor.

- **Cadenas.** Constantes de cadena (*strings*). Empiezan con una comilla doble (") y finalizan con una comilla doble ("). Todos los caracteres comprendidos entre la primera comilla y la segunda comilla se consideran el contenido del string. Además, se deben aceptar las siguientes secuencias de escape:

  - `\"` – contrabarra (\) seguida de una comilla doble ("): representa una única comilla doble (").
  - `\\` – contrabarra (\) seguida de contrabarra (\): representa una única contrabarra (\).

  Por ejemplo, la constante de cadena `"Hola \"mundo\"."` representa un texto de longitud 13.

  Idealmente las cadenas deberían ser de longitud ilimitada, pero se acepta que la implementación se restrinja a constantes numéricas de hasta 1023 caracteres.

  Ejemplos de constantes de cadena:

  ```
  ""    "a"    "b"    "abc"    "\\"    "\"a\""
  ```

- **Literales: palabras clave y símbolos reservados.** Llamamos *literales* a las palabras clave y símbolos reservados del lenguaje. El conjunto de palabras clave y símbolos reservados depende del lenguaje. Por ejemplo, en un lenguaje como Python la cadena `and` es una palabra clave (que representa el "y" lógico) y `**` es un símbolo reservado (que representa la potencia), mientras que en un lenguaje como C la palabra `and` es simplemente un identificador (que podría ser el nombre de una variable o una función) y la cadena `**` se tokeniza como una secuencia de dos símbolos reservados (es decir, dos veces el literal `*`). Es por eso que el tokenizador debe ser *paramétrico*.

  Dicho de otro modo, cuando se crea un tokenizador, se deben proveer dos conjuntos de strings:

  - Un conjunto $\mathcal{K}$ de **palabras clave**.
  - Un conjunto $\mathcal{S}$ de **símbolos reservados**.

  Por ejemplo, si el conjunto de palabras clave es vacío, es decir $\mathcal{K} = \emptyset$, y el conjunto de símbolos reservados es $\mathcal{S} = \{\texttt{"+"}\}$, entonces el siguiente archivo de entrada:

  ```
  if++x
  ```

  consta de cuatro tokens:

  1. el identificador `"if"`,
  2. el símbolo reservado `"+"`,
  3. el símbolo reservado `"+"`,
  4. el identificador `"x"`.

  En cambio si el conjunto de palabras clave es $\mathcal{K} = \{\texttt{"if"}\}$ y el conjunto de símbolos reservados es $\mathcal{S} = \{\texttt{"++"}\}$, el mismo archivo de entrada consta de sólo tres tokens:

  1. la **palabra clave** `"if"`,
  2. el símbolo reservado `"++"`,
  3. el identificador `"x"`.

  El conjunto $\mathcal{K}$ de palabras clave debe contener únicamente palabras de la forma `[a-zA-Z_][a-zA-Z0-9_]*`. Por ejemplo, las siguientes podrían ser palabras clave en algún lenguaje:

  ```
  if    then    else    while    return    BEGIN    END    _
  ```

  El conjunto $\mathcal{S}$ de símbolos reservados debe contener únicamente símbolos formados por cualquiera de los siguientes caracteres:

  ```
  ( ) [ ] { } , ; : . + - * / % ! ? $ @ # | & = < > ~ ^ \
  ```

  Por ejemplo, los siguientes podrían ser símbolos reservados en algún lenguaje:

```
+    :    ++    ->    >>    <*>    $$
```

La única excepción es que un símbolo reservado **no** puede comenzar con `/*` porque dicha combinación se utiliza para delimitar comentarios.

Las palabras clave y símbolos reservados se tratan de manera ligeramente distinta. Cuando el tokenizador reconoce una secuencia de caracteres de la forma `[a-zA-Z_][a-zA-Z0-9_]*` que podrían componer un identificador, verifica si se encuentra dentro del conjunto de palabras clave $\mathcal{K}$ o no. Por ejemplo, si $\mathcal{K} = \{\texttt{"if"}\}$, el siguiente archivo de entrada:

```
if x ifx
```

consta de tres tokens:

1. la palabra clave `"if"`.
2. el identificador `"x"`.
3. el identificador `"ifx"`.

En cambio, en el caso de los símbolos reservados, el tokenizador siempre consume el símbolo más largo que pueda consumir. Por ejemplo, si el lenguaje reconoce tanto el símbolo `+` como el símbolo `++`, es decir $\mathcal{S} = \{\texttt{"+"}, \texttt{"++"}\}$ entonces el archivo de entrada que repite cinco veces el caracter `+`:

```
+++++
```

consta de tres tokens:

1. el símbolo reservado `"++"`.
2. el símbolo reservado `"++"`.
3. el símbolo reservado `"+"`.

Para conseguir este comportamiento se recomienda ordenar los símbolos reservados de mayor a menor longitud. El tokenizador comprueba, uno por uno, si hay un prefijo de la entrada que coincida con alguno de los símbolos reservados[2].

- **Espacios en blanco.** El tokenizador debe ignorar todos los espacios (`' '`), tabs (`'\t'`), retorno de carro (`'\r'`) y fin de línea (`'\n'`).
- **Comentarios.** El tokenizador acepta comentarios comenzados con `/*` es decir, una barra `/` seguida de un asterisco `*` y finalizan cuando se encuentra la primera ocurrencia de `*/` es decir un asterisco `*` seguido de una barra `/`. Notar que el tokenizador **no** reconoce comentarios anidados.
- **Notación para los símbolos terminales.** Escribimos:
  - ⟨`identificador`⟩ para el símbolo terminal que representa un identificador.
  - ⟨`número`⟩ para el símbolo terminal que representa una constante numérica.
  - ⟨`cadena`⟩ para el símbolo terminal que representa una constante de de cadena.
  - `"..."` para los símbolos terminales que representan literales. Por ejemplo `"if"` representa la palabra clave `if` y `"++"` representa el símbolo reservado `++`.

2.1. Pseudocódigo para el tokenizador

```
function obtener_próximo_token()
    Saltear espacios y comentarios.

    C := resultado de leer el siguiente caracter de la entrada.

    Análisis de casos dependiendo del valor de C:
```

[2] La solución ideal es implementarlo con un autómata finito o un trie, pero no es necesario en el contexto de este TP.

```
    1) Si C es un dígito:
         X := resultado de leer un número de la entrada.
         Retornar un token numérico con valor X.
    2) Si C es un caracter alfabético o guión bajo:
         X := resultado de leer un identificador de la entrada.
         if X es una palabra clave {
             Retornar un token literal con valor X.
         } else {
             Retornar un token identificador con valor X.
         }
    3) Si C es una comilla:
         X := resultado de leer una cadena de la entrada.
         Retornar un token cadena con valor X.
    4) En cualquier otro caso:
         Para cada símbolo reservado X en orden decreciente de longitud {
             Si la entrada empieza con X {
                 Retornar un token literal con valor X.
             }
         }
         Informar un error de sintaxis.
end
```

3. Lectura de la gramática

3.1. Descripción informal del lenguaje Lleca

Una vez que esté programado el tokenizador, se debe programar un analizador sintáctico para reconocer la gramática escrita en el lenguaje Lleca.

OJO: no confundir este paso con el paso siguiente. En este paso únicamente se lee un archivo escrito en lenguaje Lleca y se construye una estructura de datos para representar internamente la gramática leída. Este paso **NO** es el parser genérico para analizar cualquier archivo fuente sino un parser específico para el lenguaje Lleca.

- Una **gramática** escrita en el lenguaje Lleca consta de una secuencia de 0 o más reglas.
- Cada **regla** está encabezada por un símbolo no terminal seguido de una lista de 0 o más producciones.
- Cada **producción** comienza con una barra vertical (`"|"`), seguida de la expansión (lado derecho de la producción) y seguida de una acción. La acción está dada por una flecha (`"=>"`) y un término. El término representa el árbol que resulta de hacer el análisis sintáctico de dicha producción.
- Cada **expansión** es una lista de símbolos. Los símbolos pueden ser cadenas, identificadores, o cualquiera de las palabras clave `"ID"`, `"STRING"` o `"NUM"`.
- Un **término** es una expresión recursiva que se construye de alguna de las siguientes maneras:
  - La palabra clave `"_"` (guión bajo).
  - Un identificador sin argumentos, por ejemplo, `Nil`.
  - Un identificador acompañado de una lista de argumentos encerrados entre paréntesis y delimitados por comas, por ejemplo, `f(a, b, c)`
  - Una constante de cadena.
  - Una constante numérica.
  - Una referencia a un parámetro, por ejemplo `$3`.
  - Una referencia a un parámetro seguida de un **único** argumento entre corchetes, por ejemplo `$3[f(a, b, c)]`.

El siguiente es un ejemplo sencillo de gramática:

alumnos.ll

```
base_de_alumnos
| "begin" lista_alumnos "end" => $2

lista_alumnos
|                              => Nil
| alumno ";" lista_alumnos     => Cons($1, $3)

alumno
| "#" NUM ":=" STRING          => Alumno("nombre", $4, "legajo", $2)
```

3.2. Gramática del lenguaje Lleca

Las palabras clave del lenguaje lleca son:

```
_    ID    STRING    NUM
```

Los símbolos reservados del lenguaje lleca son:

```
|    =>    $    (    )    ,    [    ]
```

A continuación se dan todas las producciones de la gramática del lenguaje Lleca. El símbolo inicial es ⟨*gramática*⟩.

⟨*gramática*⟩ ⟶ ϵ
| ⟨*regla*⟩ ⟨*gramática*⟩

⟨*regla*⟩ ⟶ ⟨identificador⟩ ⟨*producciones*⟩

⟨*producciones*⟩ ⟶ ϵ
| ⟨*producción*⟩ ⟨*producciones*⟩

⟨*producción*⟩ ⟶ "|" ⟨*expansión*⟩ "=>" ⟨*término*⟩

⟨*expansión*⟩ ⟶ ϵ
| ⟨*símbolo*⟩ ⟨*expansión*⟩

⟨*símbolo*⟩ ⟶ "ID"
| "STRING"
| "NUM"
| ⟨cadena⟩
| ⟨identificador⟩

⟨*término*⟩ ⟶ "_"
| ⟨identificador⟩ ⟨*argumentos*⟩
| ⟨cadena⟩
| ⟨número⟩
| "$" ⟨número⟩ ⟨*sustitución*⟩

⟨*argumentos*⟩ ⟶ ϵ
| "(" ⟨*lista_argumentos*⟩ ")"

⟨*lista_argumentos*⟩ ⟶ ϵ
| ⟨*término*⟩ ⟨*lista_argumentos_cont*⟩

⟨*lista_argumentos_cont*⟩ ⟶ ϵ
| "," ⟨*término*⟩ ⟨*lista_argumentos_cont*⟩

⟨*sustitución*⟩ ⟶ ϵ
| "[" ⟨*término*⟩ "]"

3.3. Analizador sintáctico para el lenguaje Lleca

Para analizar sintácticamente el lenguaje Lleca pueden implementar un parser manualmente, usando la técnica de análisis sintáctico por descenso recursivo. También pueden usar un generador de parsers para su lenguaje favorito. La gramática que se provee arriba es LL(1). Si utilizan un generador de parsers que utilice otra técnica de análisis sintáctico como LALR(1), es probable que tengan que adaptar ligeramente la gramática (por ejemplo, para eliminar la recursión a derecha).

4. Procesamiento del archivo fuente

En esta sección se describe cómo analizar sintácticamente el archivo fuente. El parser que deben implementar está basado en la técnica de análisis sintáctico LL(1).

4.1. Cálculo del conjunto de palabras clave y símbolos reservados

Una vez que se cuenta con una representación de la gramática, el primer paso es determinar cuáles son las palabras clave y símbolos reservados que usa dicha gramática. Para ello se deben recorrer los símbolos que aparezcan en las expansiones de todas las producciones. Todos los símbolos que sean cadenas representarán palabras clave o símbolos reservados.

Por ejemplo, para la gramática **robot.ll**, tenemos:

- Palabras clave: $\mathcal{K} = \{$`"AVANZAR"`, `"GIRAR"`, `"IZQ"`, `"DER"`$\}$
- Símbolos reservados: $\mathcal{S} = \emptyset$.

Por otro lado, para la gramática **alumnos.ll**, tenemos:

- Palabras clave: $\mathcal{K} = \{$`"begin"`, `"end"`$\}$
- Símbolos reservados: $\mathcal{S} = \{$`"#"`, `":="`, `";"`$\}$.

Observar que las cadenas que aparecen en las acciones (por ejemplo, `"nombre"` y `"legajo"` en el caso de **alumnos.ll**) no tienen por qué ser palabras clave ni símbolos reservados.

4.2. Construcción de la tabla de análisis sintáctico LL(1)

El cálculo de la tabla LL(1) tiene cuatro pasos, que se describen en las cuatro secciones siguientes.

4.2.1. Cálculo de símbolos anulables

El primer paso es calcular el conjunto de símbolos no terminales que son **anulables**. Un símbolo no terminal $A \in N$ es anulable cuando $A \Rightarrow^* \epsilon$. Se recuerda el pseudocódigo para calcular el conjunto `ANN` de símbolos anulables:

```
ANN := {}
Repetir hasta que no haya cambios {
  Para cada producción A -> X1 ... Xn {
    Si X1, ..., Xn son todos símbolos no terminales que están en el conjunto ANN {
      Agregar A al conjunto ANN.
    }
  }
}
```

4.2.2. Cálculo del conjunto FIRST

El siguiente paso es calcular el conjunto de primeros FIRST(X) para cada símbolo terminal o no terminal $X \in \Sigma \cup N$. Notar que este paso requiere tener ya calculado el conjunto ANN de símbolos anulables. Recordar que un símbolo terminal $a \in \Sigma$ está en el conjunto FIRST(X) cuando $X \Rightarrow^* a\beta$ para alguna cadena β. Se recuerda el pseudocódigo para calcular el conjunto FIRST(X) para cada símbolo $X \in \Sigma \cup N$:

```
Poner FIRST(x) := {x} para cada símbolo terminal x.
Poner FIRST(A) := {} para cada símbolo no terminal A.
Repetir hasta que no haya cambios {
  Para cada producción A -> X1 ... Xn {
    Para cada i = 1 .. n {
      Si X1, ..., X[i-1] son todos símbolos no terminales anulables {
        FIRST(A) := FIRST(A) U FIRST(Xi)
      }
    }
  }
}
```

4.2.3. Cálculo del conjunto FOLLOW

El siguiente paso es calcular el conjunto de **siguientes** FOLLOW(A) para cada símbolo no terminal $A \in N$. Notar que este paso requiere tener ya calculado el conjunto ANN de símbolos anulables y el conjunto FIRST(X) de primeros. Recordar que un símbolo terminal $b \in \Sigma$ está en el conjunto FOLLOW(A) cuando $S \Rightarrow^* \gamma_1 A b \gamma_2$ para ciertas cadenas γ_1, γ_2, donde S es el símbolo inicial. Se recuerda el pseudocódigo para calcular el conjunto FOLLOW(A) para cada símbolo no terminal $A \in N$:

```
Poner FOLLOW(A) := {} para cada símbolo no terminal A.
Repetir hasta que no haya cambios {
  Para cada símbolo no terminal B de cada producción A -> Y1 ... Yn B X1 ... Xm {
    Para cada i = 1 .. n {
      Si X1, ..., X[i-1] son todos símbolos no terminales anulables {
        FOLLOW(B) := FOLLOW(B) U FIRST(Xi)
      }
    }
    Si X1, ..., Xn son todos símbolos no terminales anulables {
      FOLLOW(B) := FOLLOW(B) U FOLLOW(A)
    }
  }
}
```

4.2.4. Cálculo de la tabla LL(1)

La tabla de análisis sintáctico LL(1) indica, dado un símbolo no terminal $A \in N$ y el siguiente símbolo terminal $b \in \Sigma$, cuál es la producción que debe aplicar el algoritmo. Este paso requiere tener ya calculado el conjunto ANN de símbolos anulables, el conjunto FIRST(X) de primeros, y el conjunto FOLLOW(A) de siguientes.

```
TABLA[(A, b)] := VACÍA    para cada simbolo no terminal A y cada símbolo terminal b.
Para cada producción A -> X1 ... Xn {
  Para cada i = 1 .. n {
    Si X1, ..., X[i-1] son todos símbolos no terminales anulables {
      TABLA[(A, b)] = PRODUCIR(A -> X1 ... Xn)     para cada símbolo b en FIRST(Xi).
    }
  }
  Si X1, ..., Xn son todos símbolos no terminales anulables {
    TABLA[(A, b)] = PRODUCIR(A -> X1 ... Xn)       para cada símbolo b en FOLLOW(A).
  }
}
}
```

Recordar que, si a alguna entrada le corresponden dos producciones distintas, hay un conflicto en la tabla, de tal manera que la gramática no es LL(1) y no puede aplicarse este método de análisis sintáctico. En este caso Lleca debería reportar la presencia de un conflicto en la gramática.

4.3. Análisis sintáctico

Este paso requiere que se hayan calculado el conjunto de palabras clave $\mathcal{K}$ y símbolos reservados $\mathcal{S}$ de la gramática, y que se hayan calculado los conjuntos de símbolos anulables, primeros y siguientes. El primer paso para analizar sintácticamente el archivo fuente es tokenizarlo con los conjuntos $\mathcal{K}$ y $\mathcal{S}$ ya calculados. A continuación se ejecuta el siguiente algoritmo recursivo `analizar`. Recibe como parámetro un símbolo terminal o no terminal $X \in N \cup \Sigma$. De manera recursiva devuelve un árbol (más precisamente un **término**), ejecutando las acciones tal como las especifica la gramática.

```
analizar(X) {
  Si X es un símbolo terminal {
      b := resultado de consumir un token de la entrada.
      Si (b != X) {
        Error de sintaxis (se esperaba leer X pero se encontró b).
      }
      Devolver un árbol que consta únicamente de la hoja X (con su valor asociado).
  } en caso contrario (X es un símbolo no terminal) {
      b := resultado de mirar el siguiente token de la entrada, sin consumirlo.
      Si (TABLA[(X, b)] == VACÍA) {
        Error de sintaxis (se esperaba leer X pero se encontró b).
      }
      En este caso TABLA[(X, b)] == PRODUCIR(X -> Y1 ... Yn).
      argumentos = []
      Para cada i = 1 .. n {
        argumentos.agregar(analizar(Yi))    // llamada recursiva
      }
      Devolver el árbol que resulta de aplicar la acción
        de la producción (X -> Y1 ... Yn) con la lista de
        argumentos obtenida.
  }
}
```

4.4. Acciones

El proceso de análisis sintáctico construye recursivamente un **término**. Los términos son árboles con la siguiente estructura (especificada usando un tipo de datos de Haskell):

```
data Termino = Agujero
             | Cadena String
             | Numero Int
             | Estructura String [Termino]
```

En la notación del lenguaje Lleca, el siguiente es un término posible:

```
foo(bar("hola", 123), baz(_))
```

que corresponde (en notación de Haskell) al siguiente valor:

```
Estructura "foo" [
  Estructura "bar" [
    Cadena "hola",
    Numero 123
  ],
  Estructura "baz" [
    Agujero
  ]
]
```

Se describe el significado de las acciones asociadas a cada producción:

- **Guión bajo (`_`):** construye un término `Agujero`.
- **Cadena** s**:** construye un término `Cadena` s.
  `"hola"` $\rightsquigarrow$ `Cadena "hola"`
- **Número** n**:** construye un término `Numero` n.
  `123` $\rightsquigarrow$ `Numero 123`
- **Identificador** id **sin argumentos:** construye un término `Estructura` id `[]`.
  `foo` construye el árbol `Estructura "foo" []`
- **Identificador** id **con argumentos:** construye un término `Estructura` id `[`...$argumentos$...`]`.
  `foo(bar, baz)` construye el árbol `Estructura "foo" [Estructura "bar" [], Estructura "baz" []]`
- **Parámetro** $\$n$**:** devuelve el n-ésimo argumento de la lista `argumentos` que resulta de la invocación recursiva del algoritmo `analizar`.
  `suma($1, $3)` $\rightsquigarrow$ `Estructura "suma" [rec1, rec3]`
  donde `rec1` e `rec3` representan los valores de los símbolos `Y1` e `Y3` en la producción `X -> Y1 ... Yn`.
- **Parámetro con sustitución,** $\$n[X]$ **:** el n-ésimo argumento de la lista `argumentos` es un término que resulta de la invocación recursiva del algoritmo `analizar`. Cada vez que en dicho árbol haya una ocurrencia de `Agujero` se la debe reemplazar por el valor de `X`.

  Por ejemplo, con la siguiente gramática:

```
cosa
|           =>  _
| NUM cosa =>  $2[suma(_, $1)])
```

  La entrada `30` tiene como resultado:

```
suma(_, 30)
```

  La entrada `20 30` tiene como resultado:

```
suma(suma(_, 20), 30)
```

  Y la entrada `10 20 30` tiene como resultado:

```
suma(suma(suma(_, 10), 20), 30)
```

5. 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.