Laboratorio 2 de Programación 3

Alejandro Mujica

- **Lee completamente este documento antes de comenzar a trabajar o formular preguntas.**
- **Este laboratorio es evaluado y la calificación que obtengas cuenta para tu calificación definitiva.**
- **Fechas válidas de entrega: desde el 22/03/2018 hasta el 06/04/2018.**
- **El rango de fechas de entrega no será cambiado bajo ninguna circunstancia. Toma tus precauciones al respecto.**
- **Tu entrega consiste de un archivo llamado `expnode.H`, Por favor, al inicio de este archivo, en comentario, coloquen sus nombres y números de cédula.**

1. Introducción

El objetivo de este laboratorio es que aprehendas el uso de algunas estructuras de datos de interés que nos provee el Lenguaje de Programación `C++`, además del uso y aplicación de los árboles.

Para la solución de este laboratorio debes hacer una buena revisión sobre programación orientada a objetos, sobretodo herencia, métodos virtuales y sobre escritura de métodos.

El problema que resolverás aquí, es un intérprete para un pequeño lenguaje de programación para realizar algunas operaciones aritméticas y lógicas sobre números enteros, a este lenguaje lo llamaremos PR3PL. En principio resolverás los distintos tipos de nodos para construir un árbol que represente un programa escrito en PR3PL y además, programarás una operación que, dado un programa, construya el árbol correspondiente.

Para la solución de este problema requieres tener instalada la biblioteca DeSIGNAR (su última versión puedes conseguirla en el piazza) y, opcionalmente, la biblioteca readline para que compiles y el intérprete que te es dado y juegues un poco con tu solución del problema.

2. Descripción del lenguaje

PR3PL es un lenguaje dinámicamente tipado, es decir, es al momento de ejecutar un programa cuando se evaluarán que los tipos sean los correctos.

Un programa escrito en PR3PL es una expresión. Cualquier expresión debe ser escrita de la siguiente manera: "$< nombre - expresion, param_0, \ldots, param_n >$", es decir, abres corchete angular, escribes el nombre del tipo de expresión que desees usar seguidamente de los parámetros que requiera la expresión separados por coma. Finalmente cierras el corchete angular. Nótese la analogía con los árboles representados como secuencia parentizada vistos en clases.

Las expresiones de nuestro lenguaje son las siguientes:

- `void`: Define un tipo carente de valor. Puede ser usado como "comodín" para ser retornado en evaluaciones que no requieran valores explícitos de retorno. También puede usarse para marcar el fin de una lista. La sintaxis es:

  ```
  <void>
  ```

- `int`: Define un valor entero constante. Una expresión de este tipo describe un árbol de un solo nodo, es decir, un nodo hoja en el cual se almacena el valor entero. Por ejemplo, la siguiente expresión define el valor entero constante 10.

  ```
  <int, 10>
  ```

- `isvoid`: Determina si una expresión es de tipo `void`. Una expresión de ésta, es árbol con un único sub árbol como hijo. Su forma general es

  ```
  <isvoid, e>
  ```

  donde `e` es cualquier expresión.

- `pair`: Define un tipo que almacena dos expresiones. Una expresión de este tipo, describe un árbol con dos sub árboles como hijos. Por ejemplo, la siguiente expresión define el par de constantes enteras $(10, 20)$.

  ```
  <pair, <int, 10>, <int, 20>>
  ```

  Con este tipo de expresión se pueden definir listas colocando en cada segundo elemento otro par de elementos y al final dejar un `void`. Por ejemplo, la lista $[1, 2, 3]$ podría definirse así:

  ```
  <pair, <int, 1>, <pair, <int, 2>, <pair, <int, 3>, <void>>>>
  ```

- `fst`: Define una expresión para extraer el primer valor de una expresión de tipo `pair`. Una expresión de este tipo describe un árbol con un único sub árbol como hijo. Por ejemplo, la siguiente expresión:

  ```
  <fst, <pair, <int, 10>, <int, 20>>>
  ```

  indica que, del par $(10, 20)$ se debe extraer la expresión

```
<int, 10>
```

- `snd`: Define una expresión para extraer el segundo valor de una expresión de tipo `pair`. Una expresión de este tipo describe un árbol con un único sub árbol como hijo. Por ejemplo, la siguiente expresión:

  ```
  <snd, <pair, <int, 10>, <int, 20>>>
  ```

  indica que, del par $(10, 20)$ se debe extraer la expresión

  ```
  <int, 20>
  ```

- `def`: Define una expresión para declarar una variable. Una expresión de este tipo describe un árbol con un único sub árbol como hijo y la raíz almacena el nombre de la variable declarada. Por ejemplo, la siguiente línea, declara una variable llamada "x" a la cual le asigna el valor entero 10.

  ```
  <def, x, <int, 10>>
  ```

- `var`: Define una expresión para utilizar el valor almacenado en una variable. Una expresión de este tipo describe un árbol de un único nodo, es decir, un nodo hoja en el cual se almacena el nombre de la variable. Por ejemplo una expresión `pair` puede ser definida mediante variables de la siguiente manera:

  ```
  <pair, <var, x>, <var, y>>
  ```

- `neg`: Define una expresión para negar (multiplicar por -1) otra expresión. Una expresión de este tipo se describe con un único sub árbol como hijo. Por ejemplo, la siguiente expresión niega el valor entero 10:

  ```
  <neg, <int, 10>>
  ```

- `add`: Define una expresión para sumar dos expresiones. Una expresión de este tipo describe un árbol con dos sub árboles como hijos. Por ejemplo, la siguiente expresión efectúa la suma $10 + 20$:

  ```
  <add, <int, 10>, <int, 20>>
  ```

  Esta expresión se puede utilizar combinada con `neg` para efectuar restas, por ejemplo, para efectuar la operación $10 - 20$ la expresión es la siguiente:

  ```
  <add, <int, 10>, <neg, <int, 20>>>
  ```

- `mul`: Define una expresión para multiplicar dos expresiones. Una expresión de este tipo describe un árbol con dos sub árboles como hijos. Por ejemplo, la siguiente expresión efectúa el producto $10 * 20$:

```
<mul, <int, 10>, <int, 20>>
```

- `divmod`: Define una expresión para efectuar la división de dos expresiones. Como la división no es una operación cerrada en los números enteros, entonces esta operación retorna dos valores, la división entera y el módulo de las dos expresiones. Una expresión de este tipo describe con dos sub árboles como hijos. Por ejemplo, la siguiente expresión calcula el cociente (entero) y el residuo entre 12 y 10:

  ```
  <divmod, <int, 12>, <int, 10>>
  ```

- `let`: Define una expresión que permite declarar una variable asignándole un valor determinado por una expresión. Esta definición de variable se hace para que sea utilizada en un cuerpo definido por otra expresión. Una expresión de este tipo describe un árbol con dos sub árboles como hijos. El primer parámetro de la expresión es el nombre de la variable, el segundo parámetro es la expresión que le da valor a la variable y el tercer parámetro es el cuerpo donde será utilizada la variable. Su raíz almacena el nombre de la variable declarada. Por ejemplo, la siguiente expresión declara la variable "x" a la cual le asigna el valor entero 10 y posteriormente es utilizada en la expresión que suma a "x" consigo misma:

  ```
  <let, x, <int, 10>, <add, <var, x>, <var, x>>>
  ```

- `ifgreater`: Define una expresión definida por cuatro sub expresiones (llamémoslas e_1, e_2, e_3, e_4), donde si $e_1 > e_2$, entonces se ejecutará e_3, de lo contrario se ejecutará e_4. Una expresión de esta forma describe un árbol con cuatro sub árboles como hijos. Por ejemplo, la siguiente expresión define que debe ser utilizado el valor entero 2:

  ```
  <ifgreater, <int, 10>, <int, 12>, <int, 1>, <int, 2>>
  ```

- `fun`: Define una expresión para declarar una función. Esta expresión requiere el nombre de la función, el nombre del parámetro formal y una expresión que define el cuerpo de la función. Una expresión de este tipo describe un árbol con un único sub árbol como hijo. Por ejemplo, la siguiente expresión, declara una función que toma un valor y lo eleva al cuadrado:

  ```
  <fun, pow2, x, <mul, <var, x>, <var, x>>>
  ```

- `call`: Define una expresión para llamar una función. Esta expresión requiere el nombre de la función y una expresión que define el parámetro real. Una expresión de este tipo describe un árbol con un único sub árbol como hijo. El siguiente ejemplo hace el llamado a elevar al cuadrado al entero 8.

  ```
  <call, pow2, <int, 8>>
  ```

Tenemos además una expresión llamada `closure` que no será parte de nuestros programas explícitamente, está expresión será utilizada al momento de evaluar funciones.

Existe un entorno, el cual es un conjunto de variables, definido mediante una lista enlazada de pares de elementos, donde cada par contiene un nombre de variable y una expresión asociada al nombre.

Para ejecutar un programa, se debe construir el árbol que lo modele y se debe evaluar la expresión definida por la raíz en el entorno actual. La evaluación de una expresión, retorna otra expresión resultante de efectuar las operaciones necesarias. En este lenguaje la mayoría de las expresiones retornan expresiones enteras al ser evaluadas con algunas excepciones.

Cada tipo de expresión tiene sus reglas de evaluación, las cuales son las siguientes:

- `void`: Una expresión de este tipo se evalúa a sí misma. Es decir, al ejecutar $< void >$ debe retornar una nueva expresión $< void >$.

- `int`: Una expresión de este tipo se evalúa a sí misma. Es decir, al ejecutar $< int, 10 >$ debe retornar una nueva expresión $< int, 10 >$.

- `isvoid`: Una expresión de este tipo, evalúa la expresión que la compone, si es de tipo `void`, entonces retorna una nueva expresión $< int, 1 >$, de lo contrario retorna una nueva expresión $< int, 0 >$.

- `neg`: Evalúa la expresión que la compone, si el resultado es una expresión de tipo `int`, entonces se retorna una nueva expresión de tipo `int` cuyo valor es la negación del valor en el resultado obtenido. Por ejemplo:

  ```
  <neg, <int, 10>>
  ```

  retorna una nueva expresión $< int, -10 >$. Si el tipo resultante no es un entero, entonces hay un error de tipo.

- `add`: Una expresión de este tipo se evalúa de la siguiente manera: deben evaluarse cada una de las expresiones que definen los sumandos, si el resultado de evaluar ambas expresiones son expresiones de tipo `int`, entonces se debe retornar una nueva expresión de tipo `int` cuyo valor es la suma de los dos enteros obtenidos de evaluar los sumandos. Por ejemplo:

  ```
  <add, <int, 10>, <int, 20>>
  ```

  debe evaluar cada una de las expresiones que la componen, como ambas son enteras, se evalúan a sí mismas y luego se retorna una nueva expresión entera de la forma $< int, 30 >$. Si alguna de las expresiones que componen la suma, no retorna una expresión entera al ser evaluada, entonces hay error de tipos.

- `mul`: Análoga a la suma pero aplicando multiplicación. Por ejemplo:

  ```
  <mul, <int, 10>, <int, 20>>
  ```

debe evaluar cada una de las expresiones que la componen, como ambas son enteras, se evalúan a sí mismas y luego se retorna una nueva expresión entera de la forma $< int, 200 >$.

- `divmod`: Análoga a la suma y el producto, pero se debe retornar es un tipo `pair` cuya primera expresión será la división entera entre los enteros que la componen y la segunda expresión será el residuo (módulo). Por ejemplo:

  ```
  <divmod, <int, 12>, <int, 10>>}
  ```

  retorna una expresión de la forma

  ```
  <pair, <int, 1>,<int, 2>>
  ```

- `pair`: Evalúa las dos expresiones que la componen y retorna un nuevo par con ambos resultados. Por ejemplo:

  ```
  <pair, <add, <int, 10>, <int, 20>>,<int, 200>>
  ```

  retorna

  ```
  <pair, <int, 30>,<int, 200>>
  ```

- `Fst`: Evalúa la expresión que lo compone y, si es de tipo `pair`, entonces retorna el primer elemento. En caso contrario, hay error de tipo. Por ejemplo:

  ```
  <fst, <pair, <int, 10>, <int, 20>>>
  ```

  retorna

  ```
  <int, 10>
  ```

- `Snd`: Análogo al anterior pero retorna el segundo elemento.

- `Def`: Este tipo de expresión, evalúa la expresión que la compone, se crea un par $(nombre\_variable, resultado\_expresion)$ y se agrega al entorno como una pila, retorna una nueva expresión $< void >$.

- `Var`: Busca en el entorno, aquella expresión asociada al nombre de la variable, la evalúa y retorna su resultado. Por ejemplo, si tenemos el entorno

  ```
  {("x", <int, 10>), ("y", <int, 20>), ... }
  ```

  evaluar la expresión $< var, x >$ retorna $< int, 10 >$.

- `Let`: Evalúa la expresión asignada a la variable, crea un nuevo entorno consistente en el entorno actual añadiéndole la nueva variable con su valor asociado, finalmente evalúa el cuerpo bajo el nuevo entorno y retorna lo que retorne la ejecución del cuerpo. Supongamos un entorno actual

`env1 = {("x", <int, 10>), ("y", <int, 20>)}`

y evaluamos la expresión

`<let, z, <int, 30>, <add, <var, z>, <var, x>>>`

se debe evaluar $< int, 30 >$ sobre el entorno `env1` obteniendo como resultado $< int, 30 >$, se crea un nuevo entorno

`env2 = {("z", <int, 30>), ("x", <int, 10>), ("y", <int, 20>)}`

y se evalúa el cuerpo (la suma) con el entorno `env2` dando como resultado $< int, 40 >$. Véase como la declaración de una variable local, es decir, sólo va a existir en el bloque para la cual es definida.

- `ifgreater`: Evalúa e_1 y e_2. Si alguna de las dos no da una expresión de tipo entero, entonces hay error de tipo. En caso contrario se comparan los valores enteros resultantes, si el primero es mayor que el segundo, entonces se retorna la evaluación de e_3, en caso contrario se retorna la evaluación de e_4.
- `closure`: Una expresión de este tipo se evalúa a sí misma (análoga a `int` y `void`).
- `fun`: Una expresión de este tipo crea una instancia de `closure` con el entorno en el cual se está evaluando y un clon de la función. Luego se crea un par $(nombre\_funcion, clausura)$ y se agrega al entorno como pila. Retorna un *void*.
- `call`: Esta quizás es la más complicada de todas las evaluaciones. Se debe hacer la búsqueda en el entorno de nombre de la función, el resultado de la búsqueda debe ser una clausura (`closure`), de lo contrario será un error de tipo. Luego de que se obtiene la clausura, se crea una copia del ambiente en ella, se evalúa la expresión que define el parámetro real y se crea un par $(nombre\_formal, expresion\_resultante)$ para ser agregado al nuevo ambiente. Luego se agrega a ese nuevo ambiente, el nombre de la función asociado a un clon de la clausura (para permitir llamadas recursivas) y finalmente se evalúa el cuerpo de la función con el nuevo entorno.

3. Laboratorio

Para la realización de este trabajo se te proveen los siguientes archivos:

- `helpers.H` Contiene dos funciones que sirven de ayuda para la ejecución de programas.
  - `remove_whites(str)` Recibe una cadena con un programa escrito en nuestro lenguaje y la retorna sin espacios en blancos, tabulaciones y fines de línea.

    - `is_str_num(str)` Recibe una cadena y retorna `true` si está compuesta de sólo dígitos entre 0 y 9, en caso contrario retorna `false`.

- `test.C` Un pequeño programa con algunas pruebas para que te asegures de que tus códigos son correctos. Añade a este archivo todas las pruebas que consideres necesarias.

- `interprete.C` Un pequeño programa que funge de línea de comandos para el lenguaje. Al ejecutarlo, leerá líneas, éste espera que cada línea sea una expresión del lenguaje completa, al presionar enter, la expresión es evaluada y se muestra el resultado, en caso de errores, éstos son mostrados en la línea de comandos. Para la compilación de este programa, requieres tener instalada la biblioteca readline.

- `Makefile` Para compilar el programa de pruebas y el intérprete.

- `expnode.H` Contiene la definición de los tipos de nodos para el árbol que representa el lenguaje como una plantilla para que programes la solución de la práctica. Los tipos descritos allí son los siguientes:

    - Exp Es el tipo de expresión base en nuestro lenguaje. Dentro de este se define un enumerado para el tipo de expresión que se utiliza. Estudia el código de este tipo, pero no lo modifiques. Los tipos que vas a utilizar derivan de `Exp`. Un programa en nuestro lenguaje, se conforma con un árbol de expresiones.
    - `Void` Especialización para expresiones de tipo "comodín".
    - `Int` Especialización para expresiones enteras.
    - `IsVoid` Especialización para expresiones que determinan si otra expresión es de tipo `Void` o no.
    - `Pair` Especialización para expresiones de pares de enteros.
    - `Fst` Especialización para expresiones que extraen el primer elemento de un par.
    - `Snd` Especialización para expresiones que extraen el segundo elemento de un par.
    - `Neg` Especialización para expresiones que niegan otra expresión.
    - `Def` Especialización para expresiones que declaran variables.
    - `Var` Especialización para expresiones que utilizan variables.
    - `Add` Especialización para expresiones que suman dos expresiones.
    - `Mul` Especialización para expresiones que multiplican dos expresiones.
    - `DivMod` Especialización para expresiones que dividen dos expresiones.
    - `Let` Especialización para expresiones condicionales.
    - `IfGreater` Especialización para expresiones que comparan dos expresiones para decidir cuál de otras dos se va a ejecutar.

- `Closure` Especialización para expresiones que se crean al evaluar funciones.
- `Fun` Especialización para expresiones que declaran funciones.
- `Call` Especialización para expresiones que llaman funciones.
- `Enviroment` Tipo para crear un entorno o conjunto de variables. Está basado en

  ```
  SLList<tuple<string, Exp *>>
  ```

  este tipo no lo modifiques.

Cada uno de los tipos definidos en el archivo `expnode.H` tiene su constructor y destructor ya programados y una operacion `to_string()` la cual retorna una cadena de la expresión tal como se escribe en el lenguaje. Esta operación también está programada en cada tipo (no las modifiques).

Las operaciones que tú debes programar para cada tipo son las siguientes:

- `destroy()` La cual debe liberar la memoria de las expresiones que compongan al tipo sobre el cual estás programando. Nota que en el tipo `Int` y en el tipo `Var`, este método queda vacío porque son tipos que no se componen por otras expresiones.
- `clone()` Debe crear una nueva instancia del mismo tipo con los mismos atributos y retornar su dirección. Por ejemplo el método `clone()` del tipo `Int` sería `return new Int(value);`.
- `eval(env)` Debe evaluar el tipo de expresión bajo el ambiente `env` con las reglas descritas arriba. La evaluación de una expresión, debe retornar una nueva expresión. Por ejemplo para el tipo `Int`, el método `eval(env)` es como un sinónimo de `clone()`, pues lo que hará es crear un nuevo `Int` con el mismo valor. Para otros tipos como por ejemplo `Neg`, se debe evaluar la expresión que lo compone (`e->eval(env)`) y se debe verificar si el resultado es de tipo entero (revisa los tipos en `Exp`). Si pasa la prueba de tipos, entonces se debe retornar un nuevo entero con el valor negativo. Es importante darse cuenta de que al evaluar la expresión `e`, se creó una nueva instancia del entero que fue retornado allí, por lo tanto, antes de retornar la nueva expresión, debe ser liberada la memoria de los resultados de cada evaluación hecha dentro de este método. Quizás es uno de los detalles más difíciles en esta práctica, pero pon mucha atención a eso. En caso de que falle la prueba de tipos, aparte de liberar la memoria de los resultados de las evaluaciones, debes arrojar la excepción `domain_error` con un mensaje adecuado, por ejemplo ``` ``neg applied to non-int'' ```. Para la implementación de este método en el tipo `Var`, se te provee una función denominada `envlookup` la cual recibe como parámetro el entorno y el nombre de la variable y retorna un clon de la expresión asociada al nombre.

Finalmente debes programar la función `parse()`, la cual recibe como parámetro una cadena que describe un programa, construye el árbol correspondiente a la expresión y retorna la raíz del árbol. Si la expresión está mal formada (cantidad de parámetros incorrecta para el tipo de expresión, los corchetes angulares no están bien emparejados, etc) debes arrojar una excepción de tipo `logic_error` con un mensaje de error adecuado. Es muy importante que esta rutina esté correcta y que valide bien cada expresión en cada caso. Pues toda la evaluación recae sobre ella, en la evaluación se harán pruebas de esta rutina para cada tipo y se harán pruebas de evaluación de cada tipo con los resultados obtenidos de esta rutina. Si tu rutina construye un mal árbol, entonces no pasará la prueba de `parse` y tampoco pasará las pruebas de `eval`.

4. Evaluación

La fecha de entrega de este laboratorio es desde el 22/03/2018 hasta el 06/04/2018.

Tienes permitido enviar tu práctica máximo una vez por día. Es decir, desde el día de inicio hasta el día final de la práctica tienes disponibles 5 intentos. Éstos no son acumulativos. Si un día no envías, perdiste ese intento. Si por ejemplo llegas al segundo día de la práctica y no enviaste nada el primer día, entonces te quedarían solamente 4 intentos disponibles.

Para evaluarte debes enviar el archivo `expnode.H` a la dirección:

`alejandro.j.mujic4@gmail.com`

El "subject" debe ser **exactamente** el texto **"PR3-LAB-02"** sin las comillas. Si fallas con el subject entonces probablemente tu laboratorio no será evaluado, pues el mecanismo automatizado de filtrado no podrá detectar tu trabajo. No comprimas el archivo y no envíes nada adicional a éste.

El único contenido que debe aparecer en el correo es tu número de cédula y tu nombre separados por espacio como se muestra en el siguiente ejemplo:

`V01XXXXXXX Alejandro Mujica`

Por favor, no uses otros medios para enviar la solución ni escribas otros comentarios adicionales en el email.

Atención: si tu programa no compila, entonces el evaluador no compila. Si una de tus rutinas se cae, entonces el evaluador se cae. Si una de tus rutinas cae en un lazo infinito o demora demasiado, entonces el evaluador cae en un lazo infinito o se demora demasiado. Por esa razón, en todos estos casos será imposible darte una nota, lo que simplemente se traduce en que tienes cero en el intento en el cual ocurra una de las circunstancias mencionadas.

No envíes programas que no compilan o se caen. Hacen perder los tiempos de red, de cpu, el tuyo y el mío. Si estás al tanto de que una rutina se te cae y no la puedes corregir, entonces trata de aislar la falla y disparar una excepción cuando ésta ocurra. Si no logras implementar una rutina, entonces haz que dé un valor de retorno el cual, aunque estará incorrecto y no se te evaluará, le permitirá al evaluador proseguir con otras rutinas. De este modo no tumbarás al evaluador y eventualmente éste podría

darte nota para algunos casos (o todos si tienes suerte). Si no logras aislar una falla, entonces deja la rutina tal como te fue dada, pero asegúrate de que dé un valor de retorno. De este modo, otras rutinas podrán ser evaluadas.

5. Recomendaciones

1. Lee enteramente este enunciado antes de proceder al diseño e implementación. Asegúrate de comprender bien tu diseño.

   Haz un boceto de tu estructura de datos y cómo esperas utilizarla. Por cada rutina, plantea qué es lo que vas hacer, cómo vas a resolver el problema. Esta es una situación que amerita una estrategia de diseño y desarrollo.

2. La distribución del laboratorio contiene un pequeño test. No asumas que tu implementación es correcta por el hecho de pasar el test. Construye tus casos de prueba, verifica condiciones frontera y manejo de alta escala.

3. Este es un problema en el cual los refinamientos sucesivos son aconsejables. La recomendación general es que obtengas una correcta versión operativa lo más simplemente posible. Luego, si lo prefieres, puedes optar por mejorar el rendimiento.

4. Ten cuidado con el manejo de memoria. Asegúrate de no dejar "leaks" en caso de que utilices memoria dinámica. Todo `new` que hagas debe tener su `delete` en algún lugar. valgrind **y** DDD **son buenos amigos**.

5. Usa el foro para plantear tus dudas de comprensión. Pero de ninguna manera compartas código, pues es considerado **plagio**.

6. No incluyas headers en tu archivo `expnode.H` (algo como `# include ...`), pues puedes hacer fallar la compilación. Si requieres un header especial, el cual piensas no estaría dentro del evaluador, exprésalo en el foro para así incluirlo.