Lenguaje de Programación PR3

Introducción

El objetivo de esta asignación es que aprendas y domines una de las aplicaciones de los árboles para resolver el intérprete de un lenguaje de programación declarativo denominado **PR3PL**. Si bien, en clases vimos una implementación bastante general de los árboles mediante la implementación de una clase `Node` con un apuntador a su hermano inmediato y otro apuntador a su primogénito, aquí implementarás un árbol que puede tener diferentes tipos de nodos. Cada uno de estos nodos representa una expresión de PR3PL.

Para implementar este árbol con diferentes tipos de nodos, utilizaremos el concepto de interfaz de POO. Para esto, es requerido que hagas una buena revisión sobre este concepto y de cómo se implementan en **C++** mediante clases que contienen sólo métodos virtuales puros y sobreescritura de métodos.

Intérprete de un Lenguaje de Programación

Un intérprete es un programa que ejecuta directamente instrucciones escritas en algún lenguaje de programación o de scripting sin tener que haberlas compilado previamente a un código de máquina. Generalmente, usa uno de las siguientes estrategias para la ejecución del programa:

1. Analizar gramaticalmente el código y evaluarlo directamente;
2. Traducir el código fuente a alguna representación intermedia eficiente o código objeto y ejecutarlo inmediatamente;
3. Ejecutar explícitamente bytecode precompilado generado por un compilador y combinado con la máquina virtual del intérprete.

Referencia: https://en.wikipedia.org/wiki/Interpreter_(computing)

Análisis Gramatical

El **análisis gramatical**, también conocido como **análisis sintáctico**, es el proceso de recibir símbolos de un lenguaje (natural o de programación) y verificar su correctitud según la gramática formal del lenguaje para construir un árbol de sintaxis abstracta. Estos símbolos son entregados como **tokens** al analizador sintáctico.

Un **token** es una cadena con un significado en un lenguaje. Se modela mediante un par de elementos que identifican el tipo de token y la cadena. Por ejemplo, en un lenguaje de programación convencional, podemos considerar algunos tipos de tokens como **operador**, **identificador**, **paréntesis abierto**, **paréntesis cerrado**, **punto y coma**, **número**, etc. Estos tipos dependen del significado que se le pueda dar a cada cadena en el lenguaje.

Para este proceso, es importante tener un tokenizador, es decir, un programa que reciba el flujo de caracteres que componen el programa a ser analizado y devuelva una secuencia de tokens. Un token es un par de elementos de la forma `(TokenType, Value)`. El tipo de token identifica el componente del lenguaje tal como se mencionó en el párrafo anterior. El valor es la cadena que cumple con el token; por ejemplo, para un token de tipo identificador, el valor podría ser **`x`** o para un token de tipo entero, el valor podría ser **`10`**, así podríamos tener los tokens `(Identifier, "x"), (Int, "10")`. Al tenerlos identificados de esa manera, se toman decisiones al momento de hacer el análisis sintáctico, como por ejemplo, según el contexto del identificador, saber si se debe buscar entre las variables declaradas para ser usada o declararla en ese momento. En un programa, se podría encontrar algo como `x = 10`. Un tokenizador podría producir la siguiente secuencia de tokens para la instrucción anterior: `(Identifier, "x")(AssignmentOperator, "=")(Int, "10)`. De esa manera, el analizador sintáctico sabrá que debe crear una operación de asignación.

Teniendo la secuencia de tokens, se puede verificar que estos vengan en un orden adecuado para cumplir con las reglas sintácticas del lenguaje de programación que se está implementando.

PR3PL

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

En PR3PL, todo es una expresión. Una expresión es una entidad sintáctica de un lenguaje de programación que puede ser evaluada para determinar su valor o terminar con un error. Las expresiones en este lenguaje son escritas en forma de árbol en notación parentizada, por ejemplo: `(nombre-expresión (e1) (e2) … (en))` donde e_i es una subexpresión.

Este lenguaje funciona mediante "**binding**", esto quiere decir que, en un entorno dinámico, se almacena el nombre de una variable o función asociado a una expresión. Esto no es más que un mapeo entre un identificador y una expresión resuelto por un conjunto de pares ordenados. Además, puede haber "*variable shadowing*", esto es que podemos hacer un mapeo de un identificador con una expresión y luego hacer otro mapeo del mismo identificador a otra expresión, lo que producirá que la expresión anterior quede totalmente deshabilitada para su uso (no hay errores tales como "variable previamente declarada"). Esto es debido a que cada mapeo se agrega en el entorno en forma de pila, es decir, los nuevos mapeos quedarán antes que los anteriores y al hacer una búsqueda (*lookup*) de un identificador en el entorno, se encontrará el definido más recientemente. Las variables en este lenguaje son inmutables, es decir, una vez asignado un valor, no se puede reasignar.

El léxico

Los símbolos (tipos de tokens) pertenecientes a este lenguaje de programación son los siguientes:

- **Paréntesis :** `()`
- **Literales enteros**: Cualquier número entero.
- **Identificadores**: Comienzan con una letra y solo pueden contener cualquier combinación de letras o números.
- **isunit**: palabra reservada.
- **pair**: palabra reservada.
- **fst**: palabra reservada.
- **snd**: palabra reservada.
- **- (operador de negación)**: palabra reservada.
- **+ (operador de suma)**: palabra reservada.
- *** (operador de multiplicación)**: palabra reservada.
- **/ (operador de división)**: palabra reservada.
- **% (operador de módulo o resto de división)**: palabra reservada.
- **iflesser**: palabra reservada.
- **val**: palabra reservada.
- **var**: palabra reservada.
- **let**: palabra reservada.
- **fun**: palabra reservada.
- **call**: palabra reservada.

Sintaxis

Como se menciona previamente, en PR3PL todo es una expresión. A continuación se muestra la lista de las expresiones con su significado, representación en memoria y su sintaxis:

- **unit**: Define una expresión vacía. Se representa con un árbol de un solo nodo, es decir, un nodo hoja. Su sintaxis es `()`. Esto podría ser considerado como un valor `null`.

- **int**: Define un literal entero. Se representa con un árbol de un solo nodo, es decir, un nodo hoja en el cual se almacena el valor. La sintaxis es `(valor_entero)`. Por ejemplo, el literal 10, se escribe `(10)`. Estos valores podrían representar valores booleanos con `(1)` y `(0)` respectivamente.

- **id**: Define un identificador. Se representa con un árbol de un solo nodo, es decir, un nodo hoja en el cual se almacena la cadena con el identificador. La sintaxis es `(identificador)`. Por ejemplo, el identificador x, se escribe `(x)`.

- **isunit**: Define una expresión unaria que determina si una expresión dada es de tipo **unit**. Se representa mediante un árbol con un solo sub árbol como hijo. Su sintaxis es `(isunit(e))` donde `e` es la subexpresión que será evaluada para determinar si es del

tipo requerido o no. Por ejemplo `(isunit ())` o `(isunit (10))`.

- **pair**: Define una expresión binaria para el manejo de pares ordenados. Se representa mediante un árbol con dos sub árboles como hijos. Su sintaxis es `(pair(`e_1`)(`e_2`))`. Con este tipo de expresión podrían definirse listas, usando como primer argumento una expresión y como segundo argumento otro par, el final de la lista podría ser denotado por la expresión **unit** en el segundo elemento del último par (nótese que la expresión **unit** podría representar la lista vacía); por ejemplo, si quisiéramos representar la lista de enteros [1, 2, 3] podríamos escribirla de la siguiente manera: `(pair(1)(pair(2)(pair(3)())))`. También, si quisiéramos representar la lista vacía, solo podríamos escribir `()`.

- **`fst:`** Define una expresión unaria para extraer el primer valor de un par ordenado. Se representa mediante un árbol con un único sub árbol como hijo. Su sintaxis es `(fst(e))`. Por ejemplo `(fst (pair(1)(2))`.

- **`snd:`** Define una expresión unaria para extraer el segundo valor de un par ordenado. Se representa mediante un árbol con un único sub árbol como hijo. Su sintaxis es `(snd(e))`. Por ejemplo `(snd (pair(1)(2))`.

- **neg**: Define una expresión unaria para negar (multiplicar por -1) otra expresión. Se representa con un árbol con un único sub árbol como hijo. Su sintaxis es `(-(e))`. Por ejemplo `(-(1))`.

- **add**: Define una expresión binaria para sumar dos expresiones. Se representa con un árbol con dos sub árboles como hijos. Su sintaxis es `(+(`e_1`)(`e_2`))`. Por ejemplo `(+(5)(2))`. La resta, podría representarse utilizando esta expresión combinada con la anterior. Por ejemplo: `(+(5)(-(2)))`.

- **mul**: Define una expresión binaria para multiplicar dos expresiones. Se representa con un árbol con dos sub árboles como hijos. Su sintaxis es `(*(`e_1`)(`e_2`))`. Por ejemplo `(*(5)(2))`.

- **div**: Define una expresión binaria para dividir dos expresiones. Se representa con un árbol con dos sub árboles como hijos. Su sintaxis es `(/(`e_1`)(`e_2`))`. Por ejemplo `(/(5)(2))`.

- **mod**: Define una expresión binaria para calcular el módulo de dos expresiones. Se representa con un árbol con dos sub árboles como hijos. Su sintaxis es `(%(`e_1`)(`e_2`))`. Por ejemplo `(%(5)(2))`.

- **Iflesser:** Define una expresión condicional. Se representa mediante un árbol con cuatro sub árboles como hijos. Su sintaxis es `(iflesser(`e_1`)(`e_2`)(`e_3`)(`e_4`))`. Las dos

primeras expresiones son las que se compararán, la tercera es la expresión en caso verdadero y la cuarta el caso falso. Por ejemplo `(iflesser(5)(2)(10)(pair(10)(20)))` expresa una expresión de la forma (en un lenguaje similar a SML) `if 5 < 2 then 10 else (10, 20)` . Esta expresión se puede combinar para diferentes operaciones como “mayor que”, “igual que”, entre otras. Por ejemplo, si quisiéramos preguntar si una lista está vacía o no para, y evaluar `true` en caso de que sí y `false` en caso contrario, podríamos escribirla de la siguiente manera `(iflesser(0)(isunit())(1)(0))`, esto puede leerse como “si cero es menor que el resultado de evaluar si la expresión `()` es unit, entonces evalúe en `1`, sino en `0`. Este uso es redundante, similar a preguntar que si algo es verdadero retorne verdadero y en caso contrario retorne falso. Hay ejemplos más elaborados que se te proveen.

- **val**: Define una expresión binaria para crear un *binding* de una variable. Se representa con un árbol con dos sub árboles como hijos. El primer hijo es la expresión con el identificador y el segundo es la expresión que se le asociará. Su sintaxis es `(val(`e_1`)(`e_2`))`. Por ejemplo `(val(x)(10))`.

- **var**: Define una expresión unaria para usar la expresión asociada a una variable. Se representa con un árbol con un único sub árbol como hijo, este contiene la expresión con el identificador. Su sintaxis es `(var(e))`. Por ejemplo `(var(x))`.

- **let**: Define una expresión que permite hacer un *binding* “local” de una variable para ser usada en un cuerpo dado. Se representa con un árbol con tres sub árboles como hijos. El primer hijo es la expresión con el identificador, el segundo hijo es la expresión que se le asigna a la variable y el tercero es el cuerpo en el cual se utilizará la variable. Su sintaxis es: `(let(`e_1`)(`e_2`)(`e_3`))`. Por ejemplo:
  `(let(x)(10)(pair(var(x))(20)))`.

- **fun**: Define una expresión para crear un *binding* de una función. Una función en el sentido más puro, recibe un único parámetro. Si se requieren varios parámetros, se pueden usar pares ordenados para conformar listas de paŕametros para pasarlos juntos en uno solo. Se representa mediante un árbol con tres sub árboles como hijos. El primero de los hijos es el identificador de la función, el segundo el identificador del parámetro formal y el tercero es el cuerpo de la función. Su sintaxis es `(fun(`e_1`)(`e_2`)(`e_3`))`. Por ejemplo, la función para elevar un entero al cuadrado, podría definirse como sigue: `(fun(pow2)(x)(*(var(x))(var(x))))`.

- **call**: Define una expresión para llamar una función. Se representa con un árbol con dos sub árboles como hijos que representarán el identificador de la función y la expresión que se le pasará a la función como argumento. Su sintaxis es: `(call(`e_1`)(`e_2`))`. Por ejemplo, para elevar el entero 8 al cuadrado, se puede utilizar la función del ejemplo

previo de la siguiente manera: `(call(pow2)(8))`.

Además, tenemos una expresión llamada **closure** (clausura) que no será parte de nuestros programas explícitamente. Está expresión se crea al momento de definir una función de manera tal que estas operen bajo en el entorno en el cual fue definida y no en el que sea llamada y así, poder usar dentro de la función cualquier mapeo definido en el entorno al momento de crearla. Si ocurre un *shadowing* de posterior de algún identificador usado dentro de la función, su comportamiento no se verá afectado.

Existe un entorno definido mediante una lista enlazada de pares de elementos, donde cada par contiene un identificador y una expresión asociada.

Para ejecutar un programa, se debe construir el **árbol de sintaxis abstracta** 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. La semántica del lenguaje define cómo se evalúa cada una de las expresiones.

Semántica

Cada una de las expresiones descritas arriba, se deben evaluar de la siguiente manera:

- **unit**: Una expresión de este tipo se evalúa a sí misma. Es decir, al evaluarse, debe retornar una nueva expresión de tipo **unit**.

- **int**: Una expresión de este tipo se evalúa a sí misma. Es decir, al evaluarse, debe retornar una nueva expresión de tipo **int** con el mismo valor.

- **id**: Una expresión de este tipo se evalúa a sí misma. Es decir, al evaluarse, debe retornar una nueva expresión de tipo **id** con el mismo valor.

- **isunit**: Evalúa la expresión hija con el entorno recibido, si el resultado de evaluarla es de tipo **unit**, entonces retorna una nueva expresión **int** con el valor 1, de lo contrario retorna una nueva expresión **int** con el valor 0.

- **pair**: Evalúa sus dos expresiones hijas con el entorno recibido y retorna una nueva expresión **pair** compuestas por los resultados obtenidos.

- **`fst:`** Evalúa la expresión hija con el entorno recibido, si esta es de tipo par, entonces retorna el primer hijo del par resultante, en caso contrario, se debe arrojar la excepción `TypeError` con el mensaje “fst applied to non-pair”.

- **`snd:`** Evalúa la expresión hija con el entorno recibido, si esta es de tipo par, entonces retorna el segundo hijo del par, en caso contrario, se debe arrojar la excepción

`TypeError` con el mensaje “snd applied to non-pair”.

- **neg**: Evalúa la expresión hija con el entorno recibido, si la expresión resultante es de tipo **int**, entonces retorna una nueva expresión **int** con el valor de la expresión resultante negado. Si la expresión resultante es de cualquier otro tipo, entonces se debe arrojar la excepción `TypeError` con el mensaje “negation applied to non-int”.

- **add**: Evalúa las dos expresiones hijas con el entorno recibido, si ambos resultados son de tipo **int**, entonces se retorna una nueva expresión **int** cuyo valor será la suma de los valores almacenados en ambos resultados. En caso contrario, se debe arrojar la excepción `TypeError` con el mensaje “addition applied to non-int”.

- **mul**: Evalúa las dos expresiones hijas con el entorno recibido, si ambos resultados son de tipo **int**, entonces se retorna una nueva expresión **int** cuyo valor será el producto de los valores almacenados en ambos resultados. En caso contrario, se debe arrojar la excepción `TypeError` con el mensaje “multiplication applied to non-int”.

- **div**: Evalúa las dos expresiones hijas con el entorno recibido, si ambos resultados son de tipo **int**, entonces se retorna una nueva expresión **int** cuyo valor será la división entera de los valores almacenados en ambos resultados. En caso contrario, se debe arrojar la excepción `TypeError` con el mensaje “division applied to non-int”. Además, si el denominador es cero, debe arrojar la excepción `OperationError` con el mensaje “division by zero”.

- **mod**: Evalúa las dos expresiones hijas con el entorno recibido, si ambos resultados son de tipo **int**, entonces se retorna una nueva expresión **int** cuyo valor será resto de dividir los valores almacenados en ambos resultados. En caso contrario, se debe arrojar la excepción `TypeError` con el mensaje “module applied to non-int”. Además, si el denominador es cero, debe arrojar la excepción `OperationError` con el mensaje “division by zero”.

- **Iflesser:** Evalúa las dos primeras expresiones hijas con el entorno recibido. Si alguna de ellas no es de tipo **int**, entonces se debe arrojar la excepción `TypeError` con el mensaje “iflesser comparison applied to non-int”. En caso contrario, se comparan los valores almacenados en los resultados. Si el primero es menor que el segundo, entonces se retorna el resultado de evaluar la tercera expresión en el entorno actual, en caso contrario, se retorna el resultado de evaluar la cuarta expresión en el entorno actual.

- **val**: Evalúa la expresión del identificador en el entorno recibido, si el resultado no es de tipo **id**, se debe arrojar la excepción `TypeError` con el mensaje "val applied to non-identifier". En caso contrario, evalúa la segunda expresión con el entorno recibido, agrega al inicio del entorno un par el nombre de la variable asociado con el resultado de

la evaluación previa y retorna una nueva expresión de tipo **unit**.

- **var**: Evalúa la expresión hija en el entorno recibido, si el resultado no es de tipo **id**, se debe arrojar la excepción `TypeError` con el mensaje "var applied to non-identifier". En caso contrario, busca la expresión asociada al nombre de la variable en el entorno recibido y la retorna. En caso de que no exista, debe arrojar la excepción `NotFoundError` con el mensaje “variable v does not exist” donde v es el nombre de la variable.

- **let**: Evalúa la expresión del identificador en el entorno recibido, si el resultado no es de tipo **id**, se debe arrojar la excepción `TypeError` con el mensaje "let applied to non-identifier". En caso contrario, evalúa la expresión se le asignó a la variable en el entorno recibido, crea un nuevo entorno consistente en el entorno actual (una copia) añadiendo al principio la nueva variable asociada al resultado de la evaluación anterior, finalmente evalúa la expresión hija que representa el cuerpo con el nuevo entorno y retorna la expresión resultante.

- **fun**: Evalúa la expresión del identificador de la función y la del nombre del parámetro en el entorno recibido, si el resultado de alguna de estas no es de tipo **id**, se debe arrojar la excepción `TypeError` con el mensaje "fun applied to non-identifier". En caso contrario, crea una nueva expresión de tipo **closure** asignándole el entorno recibido y una nueva expresión de tipo **fun** con los datos de la que se está evaluando (es decir, una copia de esta), luego añade al inicio del entorno recibido el nombre de la función asociado a la expresión de tipo **closure** creado y retorna una nueva expresión de tipo **unit**.

- **call**: Esta quizás es la más compleja de todas las evaluaciones. Evalúa la expresión del nombre de la función en el entorno recibido, si el resultado no es de tipo **id**, se debe arrojar la excepción `TypeError` con el mensaje "call applied to non-identifier". En caso contrario, se debe hacer la búsqueda en el entorno del nombre de la función, el resultado de la búsqueda debe ser de tipo **closure**. En caso de que no exista una expresión asociada al nombre de la función, se debe arrojar la excepción `NotFoundError` con el mensaje “function fn does not exist” donde fn es el nombre de la función. En caso de que sí exista una expresión asociada al nombre de la función, pero que esta no sea de tipo **closure**, se debe arrojar la excepción `TypeError` con el mensaje “call applied to non-closure”. En caso de que sí sea de tipo **closure**, se crea una copia del entorno almacenado en la clausura y se añade al inicio el nombre del parámetro de la función asociado al resultado de evaluar la expresión pasada como argumento. Luego se agrega a ese nuevo ambiente el nombre de la función asociada a la clausura (para permitir llamados recursivos, recuerda el concepto de apilar los llamados) asociada a la función almacenada en esta y, finalmente, se evalúa el cuerpo de la función con el nuevo entorno. Se debe retornar la expresión resultante de la evaluación.

Laboratorio

Para la realización de este trabajo, se te entrega la siguiente distribución de código fuente:

```
├── lib
│   ├── include
│   │   ├── Exceptions.hpp
│   │   ├── PR3PL.hpp
│   │   ├── Scanner.hpp
│   │   ├── Token.hpp
│   │   └── Types.hpp
│   ├── obj
│   └── src
│       ├── Exceptions.cpp
│       ├── PR3PL.cpp
│       ├── Scanner.cpp
│       ├── Token.cpp
│       └── Types.cpp
├── main
│   ├── bin
│   └── src
│       ├── DemoCountDownList.cpp
│       ├── DemoCountList.cpp
│       ├── DemoEval.cpp
│       ├── DemoFactorial.cpp
│       ├── DemoFibonacci.cpp
│       ├── DemoParser.cpp
│       ├── DemoQuicksort.cpp
│       ├── DemoScanner.cpp
│       ├── DemoSumList.cpp
│       └── PR3PL-CLI.cpp
├── Makefile
└── test
    ├── bin
    └── src
        └── PR3PL.cpp
```

Los archivos **`Scanner.hpp`** y **`Scanner.cpp`** contienen la implementación de un tokenizador. Este te servirá para resolver parte de la asignación. El programa **`DemoScanner.cpp`** contiene un ejemplo de cómo usarlo. **Estudia estos archivos pero no los modifiques**.

Los archivos **`Token.hpp`** y **`Token.cpp`** contienen los tipos de tokens que se manejan en el lenguaje. **Estudia estos archivos pero no los modifiques**.

Los archivos **`Types.hpp`** y **`Types.cpp`** contienen tipos básicos de estructuras de datos que te servirán para la solución de tu trabajo. Se define la clase **`Environment`** como una lista simplemente enlazada que almacena identificadores asociados a expresiones. Esto para crear variables y funciones, te provee métodos para añadir mapeos al inicio de la lista y para buscar la primera ocurrencia de un identificador y retornar la expresión asociada, en caso de no existir el identificador, retorna `nullptr`. También está la clase **`Expression`**, esto es una interfaz para todas las expresiones del lenguaje. Como adicional, están las clases **`UnaryExpression`** y **`BinaryExpression`**, ambas derivan de `Expression` e contienen una expresión hija y dos expresiones hijas (izquierda y derecha) respectivamente con sus *getters*. Estas están destinadas para ser derivadas por aquellas expresiones que requieren al menos una expresión hija o las que requieren al menos dos expresiones hijas, se define también la expresión unaria **`Closure`** para ser cread al definir una función. Aquí también se provee la macro **`LOG`**, la cual recibe los mismos argumentos que la función **`printf`** para escribir mensajes con formatos. Los mensajes escritos con esta macro solo los podrás ver en tus pruebas y no contaminarán la salida del evaluador. Además, es una buena práctica en el desarrollo de Software el uso de loggers. **Estudia estos archivos pero no los modifiques**.

La carpeta **`main/src`** contiene una serie de programas que demuestran algunas funcionalidades del lenguaje. Usa el comando **`make main`** para compilarlos y los ejecutables quedarán en la carpeta **`main/bin`**. Particularmente, el programa **PR3PL-CLI.cpp** requiere que tengas instalada la biblioteca libreadline. Si no la quieres instalar, entonces elimina ese archivo y modifica el `Makefile` para que no enlace esa biblioteca. Es solo una línea de comandos interactiva para este lenguaje de programación.

En la carpeta **`test/src`** encontrarás un programa llamado **`Test.cpp`** que contiene algunos casos de prueba básicos para tus soluciones, agrega todas las que consideres necesarias. Lo puedes compilar con **`make test`** y el ejecutable quedará en la carpeta **`test/bin`**.

`PR3PL.hpp` contiene las declaraciones de las expresiones que debes resolver y **`PR3PL.cpp`** contiene las implementaciones vacías. Aquí es donde debes efectuar tu trabajo.

Lo primero que deberías hacer, es programar el método **`eval`** de cada una de las expresiones según la semántica descrita en la sección anterior. Las expresiones están definidas mediante clases como se especifica a continuación:

- **Unit**: Deriva directamente de **Expression**.
- **Int**: Deriva directamente te **Expression**.
- **Id**: Deriva directamente te **Expression**.
- **IsUnit**: Deriva de **UnaryExpression**, la expresión hija está heredada de la clase base.
- **Pair**: Deriva de **BinaryExpression**. El primer elemento del par está dado por la expresión izquierda de la clase base y el segundo elemento por la expresión derecha de la clase base.
- **Fst**: Deriva de **UnaryExpression**, la expresión hija está heredada de la clase base.
- **Snd**: Deriva de **UnaryExpression**, la expresión hija está heredada de la clase base.

- **Neg**: Deriva de **UnaryExpression**, la expresión hija está heredada de la clase base.
- **Add**, **Mul**, **Div**, **Mod**: Derivan de **BinaryExpression**. El primer operando de cada una está dado por la expresión izquierda de la clase base y el segundo operando está dado por la expresión derecha de la clase base.
- **IfLesser**: Deriva de **BinaryExpression**. Las dos expresiones que se van a comparar están dadas por las expresiones de la clase base, esta se extiende dos las expresiones **true_expression** y **false_expression** que serán evaluadas según la expresión izquierda sea menor que la expresión derecha.
- **Val**: Deriva de **BinaryExpression**. La expresión del identificador es la izquierda de la clase base y la del cuerpo asociado es la derecha de la clase base.
- **Var**: Deriva de **UnaryExpression**. La expresión del identificador está dada por la expresión de la clase base.
- **Let**: Deriva de **BinaryExpression**. La expresión izquierda de la clase base contiene el identificador de la variable y la expresión derecha de la clase base contiene el cuerpo la expresión que se asocia a la variable. Se extiende con una expresión **body_expression** que contiene el cuerpo en el que se usará la variable.
- **Fun**: Deriva de **UnaryExpression**. La expresión de la clase base contiene el cuerpo de la función. Se extiende con las expresiones **function_name** y **parameter_name**.
- **Call**: Deriva de **BinaryExpressión**. La expresión izquierda de la clase base contiene la expresión del argumento que se le pasa a la función, la expresión derecha de la clase base contiene la clausura.

También está la clase **`Parser`**. En esta debes implementar el método de clase (marcado como `static`) llamado **`parse`**, el cual recibe una cadena con una expresión en su notación parentizada y debe retornar la raíz del árbol de sintaxis resultante. En caso de que haya algún error en la sintaxis, entonces se debe arrojar la excepción `BadProgram` con el mensaje que consideres más apropiado. Tenga en cuenta que un programa solo consistente en una expresión **id** no es válido.

Evaluación

- La práctica debe ser resuelta individualmente. Se penalizará por plagio.
- Puedes realizar preguntas por el chat de la práctica si no implica compartir código.
- Tienes dos (2) semanas para enviar la solución.
- Puedes entregar cinco (5) veces.
- Solo puedes hacer una entrega por día.
- El estilo de código tiene un peso del 10% de la calificación de la evaluación.
- Los archivos que entregues deben tener tu nombre, apellido y cédula como comentario al inicio.
- No se permite usar ningún statement de impresión por salida estándar. Para eso, usa la macro `LOG`.
- Si no cumples con las reglas anteriores, no será procesada la evaluación y perderás una entrega.

Para evaluarte, debes enviar un correo con el archivo **`PR3PL.cpp`** (**NO COMPRIMIDO)** como adjunto a la dirección alejandro.j.mujic4@gmail.com con el asunto: **[PR3]-02-PR3PL**. El cuerpo del correo debe tener tu nombre y número de cédula.

Deadline para resolver problemas de compilación: Lunes 15/07/2024 21:59:59 VET
Deadline para entregas: Viernes 26/07/2024 23:59:59 VET