TP 2: Parallel Lisp

Taller de Programación

Agustina Barbetta 96528

12 de Abril
1er. Cuatrimestre 2016

1. Introducción

Este trabajo práctico tiene por objetivo la implementación de una versión simplificada de un intérprete de Lisp, con el agregado de ejecuciones paralelas que permitan mejorar la performance general.

2. Implementación

2.1. LispInterpreter

Esta clase encapsula y abstrae la lógica del intérprete paralelo, contiene un contexto de clase `Context Protected` y un vector de `ExpressionThread*` que será completado por el `ExpressionThreadParser` a medida que va procesando las lineas del archivo entrante por `stdin`. Este objeto posee un único método que ejecuta el intérprete, en el mismo se instancia un objeto `ExpressionThreadParser` al que se le da una linea leída del archivo entrante, además de una referencia al vector de `ExpressionThread*` y el `ContextProtected`. Se evalúa el resultado del parseo, el cual puede ser:

- `EMPTY_INPUT`: En cuyo caso se continua con el ciclo, leyendo la próxima línea del archivo.
- `INVALID_INPUT_ERROR`: En cuyo caso se imprime la línea leída y se termina el ciclo devolviendo el código de error 2.
- `SYNC`: En cuyo caso se hace un *join* de todos los hilos contenidos en el vector y se continua con el parseo.

Si el resultado es distinto a las contantes anteriores, el último hilo del vector (el de la expresión que representa a la última línea parseada) se lanza. Además contiene dos métodos privados, `join_threads()` y `delete_threads()`. El método `delete_threads()` recorre el vector de hilos liberando cada uno. El método `join_threads()` recorre el vector de hilos para realizar un *join()* sobre cada uno, elimina los hilos por medio del método anterior e instancia un nuevo vector de `ExpressionThread*`.
El *join* final, se realiza en el destructor del objeto `LispInterpreter` junto con la liberación de los hilos.

2.2. Parser

Según el diseño del presente trabajo práctico, hay dos instancias en las que se requiere de parseo, por lo que se implementaron los *functors* `ExpressionThreadParser` y `ExpressionParser`. Los mismos comparten parte de funcionalidad pero no su interfaz, es por esto que se decidió hacer uso de una composición. Ambos objetos contienen un objeto `Parser` como atributo.
La clase `Parser` contiene como atributo un `map<std::string, LispFunctionFactory*>` y los siguientes métodos:

- `valid_line`: Recibe una línea y devuelve `true` si el balance de paréntesis es correcto, `false` en caso contrario.
- `get_function_name`: Recibe la línea completa leída del archivo y devuelve una *string* con el nombre de la función más abarcativa.

- `has_function`: Recibe una *string* con el nombre de la función más abarcativa, devuelve `true` si corresponde a un objeto hijo de `LispFunction` cuya *factory* existe en el *map*, `false` en caso contrario.
- `get_function`: Recibe una *string* con el nombre de la función más abarcativa, devuelve una nueva instancia de la `LispFunction` correspondiente por medio del uso de la `LispFunctionFactory` correspondiente.
- `get_arguments`: Recibe la línea completa leída del archivo y devuelve un arreglo de `Argument*`. Esta función recorre la línea desde el principio hasta el final de los argumentos, reconociendo cada uno de ellos según sus delimitadores e instanciando el hijo correspondiente de `Argument` (Los cuales pueden ser `Integer`, `String`, `Symbol` o `Function`).

El *functor* `ExpressionThreadParser` sobrecarga el operador `()`, recibiendo una línea leída del archivo entrante, una referencia a un vector de `ExpressionThread*` y otra al `ContextProtected` del intérprete. Utiliza su atributo `Parser` para realizar el parseo, instanciando el `ExpressionThread` correspondiente y agregándolo al vector recibido. Se devuelve un entero que indica si la línea leída es vacía (es decir, no se agregó un nuevo `ExpressionThread` al vector), si la expresión de la línea es un (sync) o si la operación de parseo fue normal y exitosa.
El *functor* `ExpressionParser` sobrecarga el operador `()`, recibiendo una *string* que contiene una expresión de Lisp. Utiliza su atributo `Parser` para realizar el parseo, instanciando la `Expression` correspondiente y devolviendo una referencia a ella.

2.3. Argument

Esta clase abstracta representa a los objetos `String`, `Integer`, `Symbol` y `Function`, los cuales son los cuatro argumentos posibles que puede tener una expresión (Las listas son consideradas como funciones). El único método a redefinir de esta clase es `Element* to_element(ContextProtected& context)`. Para los argumentos `String` e `Integer` la redefinición es simple, pues ellos mismos son elementos, por lo tanto, devuelven una copia de sí mismos.
En el caso de `Function`, para obtener el elemento que representa, se debe parsear la expresión guardada en la *string* `sentence` que tiene como único atributo por medio de un `ExpressionParser` y evaluarla, obteniendo un ambiente que contendrá un único elemento, el resultado.
Por último, en el caso de `Symbol` se hace uso del *context* recibido. Si el identificador del símbolo se encuentra en el *context*, se devuelve una copia del elemento correspondiente que hay en el mismo. Si el identificador no corresponde a ningún elemento del *context*, la operación debería ser un **setq** por lo que se devuelve un objeto `String` con su identificador como cadena para que la expresión se lleve a cabo.

2.4. Element

La clase abstracta `Element` define las operaciones que pueden realizarse con los elementos de Lisp. Sus hijos son las clases `String` (cuyo único atributo es una `std::string`), `Integer` (cuyo único atributo es un `int`), y `List` (cuyo único atributo es un vector de `Element*`). Estos elementos saben sumarse, restarse, multiplicarse, mostrarse en pantalla, etc. Pudiendo así ejecutar todas las operaciones pedidas para el intérprete de Lisp.

2.5. Environment

Esta clase representa a un conjunto de objetos `Element`. La misma tiene como atributo a un vector de estos objetos y métodos para su guardado y acceso. Cada expresión a evaluar creará un ambiente, transformando los argumentos recibidos en elementos, ejecutará la función y devolverá un ambiente nuevo con el elemento resultante.

2.6. Expression

Esta familia representa a las expresiones de Lisp. La clase abstracta `Expression` abarca a todos los tipos de expresión que pueden encontrarse en el programa y contiene el método virtual puro `Environment* evaluate(ContextProtected& context)`.
De ella heredan las clases `BasicExpression` y `ConditionalExpression`, estas son las expresiones que

se pueden encontrar anidadas dentro de una mayor y no se ejecutarán en hilos separados. En el caso de `BasicExpression`, sus atributos son una referencia a su correspondiente `LispFunction` y un arreglo de `Argument*`, ambos otorgados por el `ExpressionParser` al momento de su creación. Mientras que `ConditionalExpression` posee tres `Argument*`, también otorgados por el parser, de los cuales uno nunca se convertirá en `Element` dependiendo la verdad o falsedad del elemento condición.
Para representar a las expresiones más generales que se evaluarán en hilos paralelos se creó la clase `ExpressionThread` la cual contiene la lógica de lanzamiento y unión de hilos además del un context que necesitará para pasar por parámetro a la función dentro de una *callback* virtual pura, que será `evaluate(ContextProtected& context)` de su padre `Expression`.
Las expresiones que se pueden ejecutar en hilos paralelos son `BasicExpressionThread`; heredera de `ExpressionThread` y `BasicExpression`, `ConditionalExpressionThread`; heredera de `ExpressionThread` y `ConditionalExpression`, y `SetqExpressionThread`; heredera `ExpressionThread` y `Expression` (notar que la expresión **setq** nunca puede estar anidada en una mayor, ya que no tiene un elemento resultante). Todas ellas redefinen el método virtual puro `void run()` que será llamado por el método de clase `runner` de `ExpressionThread` (*callback* de `pthread_create(...)`). Cabe aclarar que, en el caso de `SetqExpressionThread` también se redefine el método `evaluate(...)` de `Expression` en el cual se utilizan los atributos *identifier* y *variable* provistos por el `ExpressionThreadParser` que referencian a objetos de clase `Argument`.

2.7. LispFunction

Por medio de esta clase se abstrae el comportamiento de todas las funciones básicas de Lisp +, -, *, /, =, <, >, list, append, car, cdr, defun y print. Las clases hijas redefinirán el método `Element* run(std::vector<Element*>& parameters)`. Cada una de ellas sabe manipular los elementos que le llegan para realizar su operación y obtener el resultado que devuelve.

2.8. LispFunctionFactory

Es una clase abstracta con un único método `LispFunction* get_function()`, de la cual heredarán múltiples fábricas de funciones de Lisp que devolverán una nueva instancia de las mismas con el llamado a `get_function()`.

2.9. Context

Esta clase representa al conjunto de variables globales definidas por medio de la operación **setq**. Posee un único atributo `map<std::string, Element*>` (Cuya clave será la cadena que identifica al elemento global y su valor la referencia al mismo) y métodos para el acceso y modificación del mismo. Cabe aclarar que cuando se intenta acceder a un elemento, el contexto entrega un clon del mismo por medio del método `clone()` y los constructores por copia de las clases hijas de `Element`.
Como cada línea del archivo entrante se convertirá en un hilo ejecutándose paralelamente a todos los demás, se nos pidió proteger las zonas críticas de nuestro sistema con **mutex**. El objeto context que tendrá como atributo la instancia de `LispInterpreter` creada en el *main* será el mismo para todos los hilos, este es el recurso que comparten y se debe proteger.
Para ello, se implementaron las clases `Mutex`; que encapsula toda la lógica de creación, bloqueo, desbloqueo y destrucción del único mutex del programa, `Lock`; que encapsula el bloqueo y desbloqueo del mutex en su constructor y destructor, y la clase `ContextProtected`; compuesta por un objeto `Mutex` y un objeto `Context`. Esta última define todos los métodos del context, llama al mismo y retorna su resultado, pero antes de hacerlo, intancia un objeto de clase `Lock` y le pasa una referencia del mutex. De esta forma, el mutex se bloquea en ese momento y se desbloquea al terminar el scope de la función.

2.10. main

Se implementó una función `main` que instancia un `LispInterpreter` y llama al método ejecutar.
Como pide el enunciado, se realiza un simple manejo de errores sobre los argumentos recibidos al ejecutar. Se verifica que la cantidad sea nula, en caso contrario se devuelve 1. Además, en el caso de que alguna línea del archivo recibido sea inválida, se devuelve 2. Para ello se utilizan las constantes `EXIT_FAILURE_1`, `EXIT_FAILURE_2` y `EXIT_SUCCESS`, las cuales corresponden a estos valores.

3. Diagramas

3.1. Clases

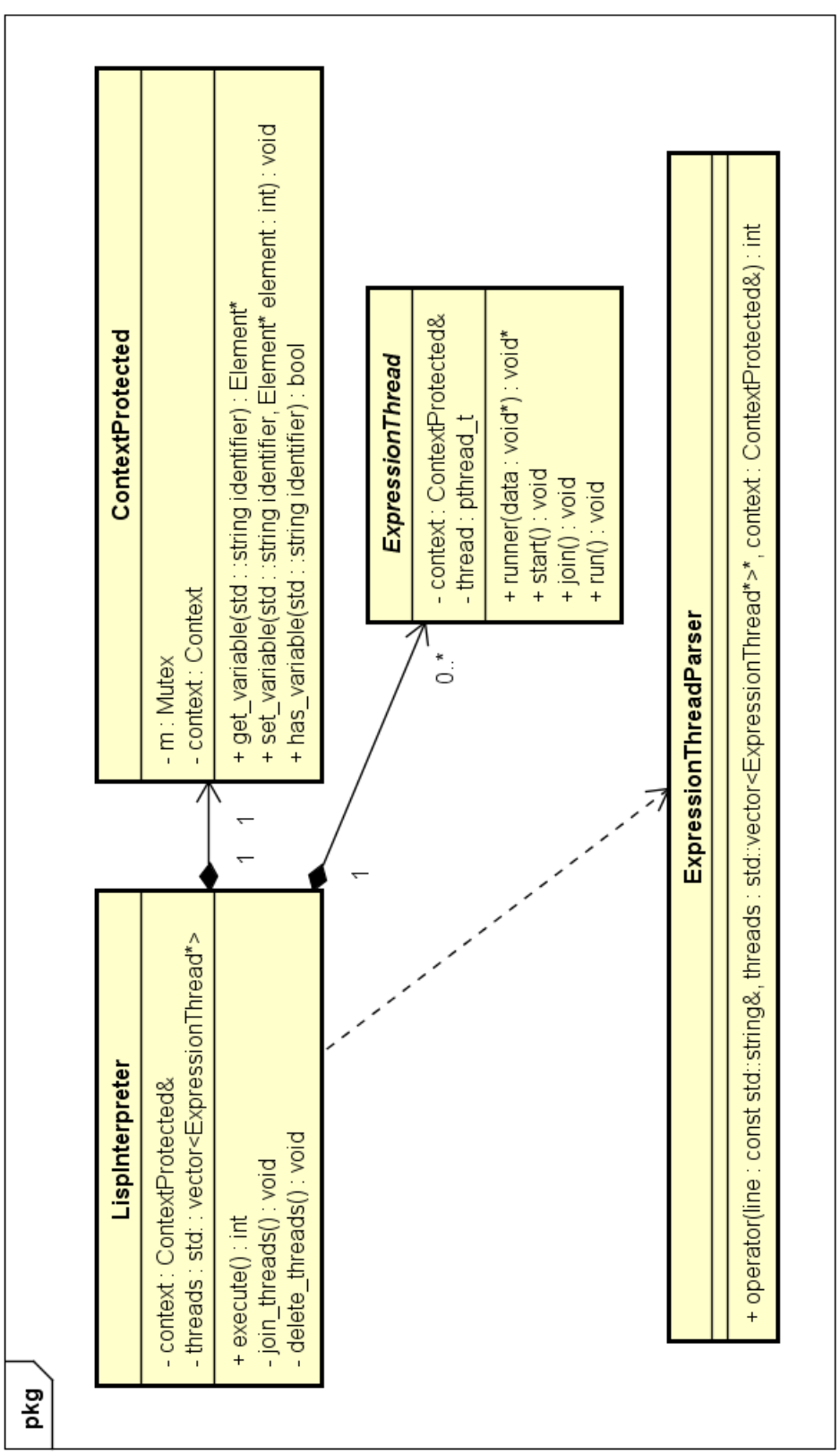

Figura 1: LispInterpreter

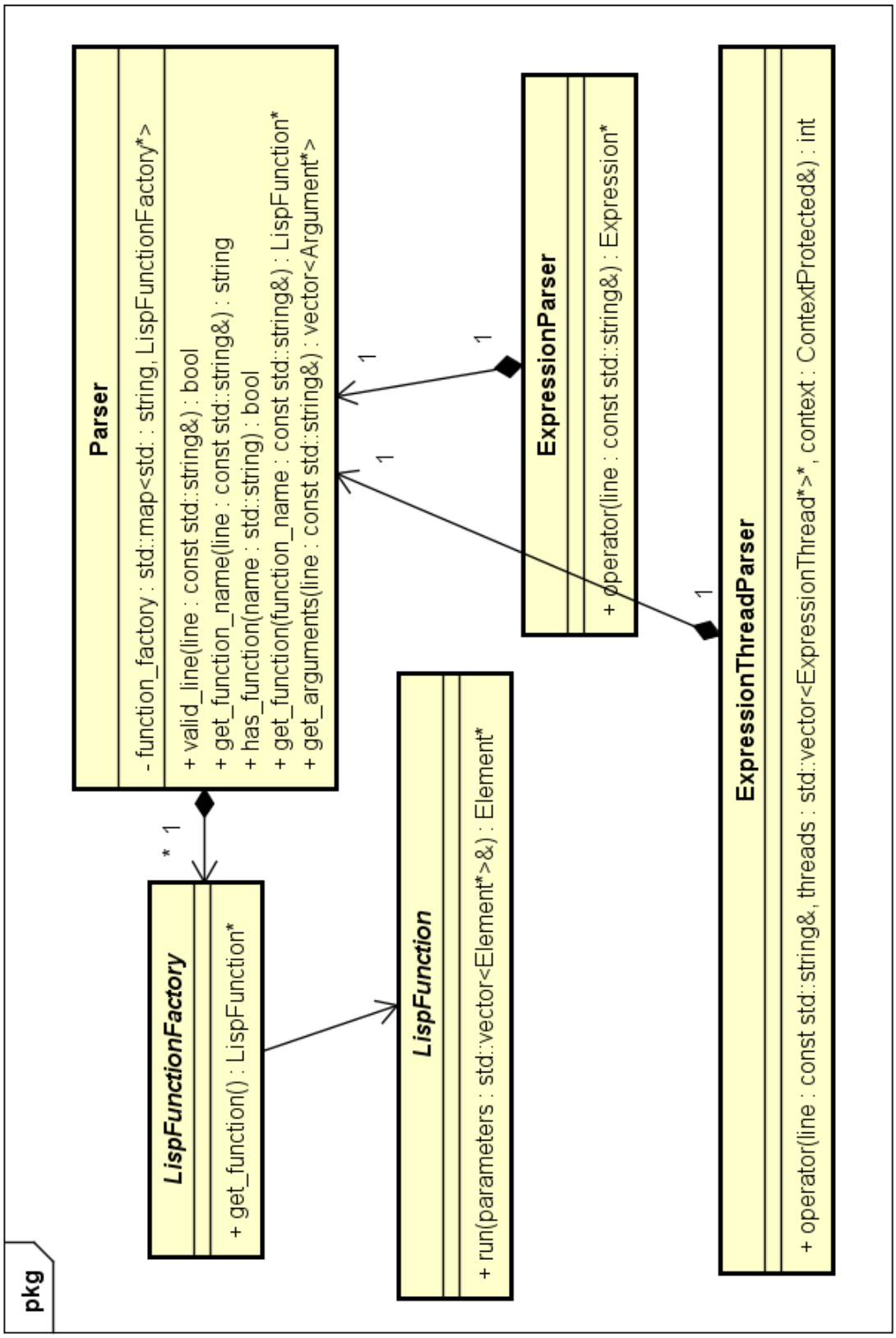

Figura 2: Parser

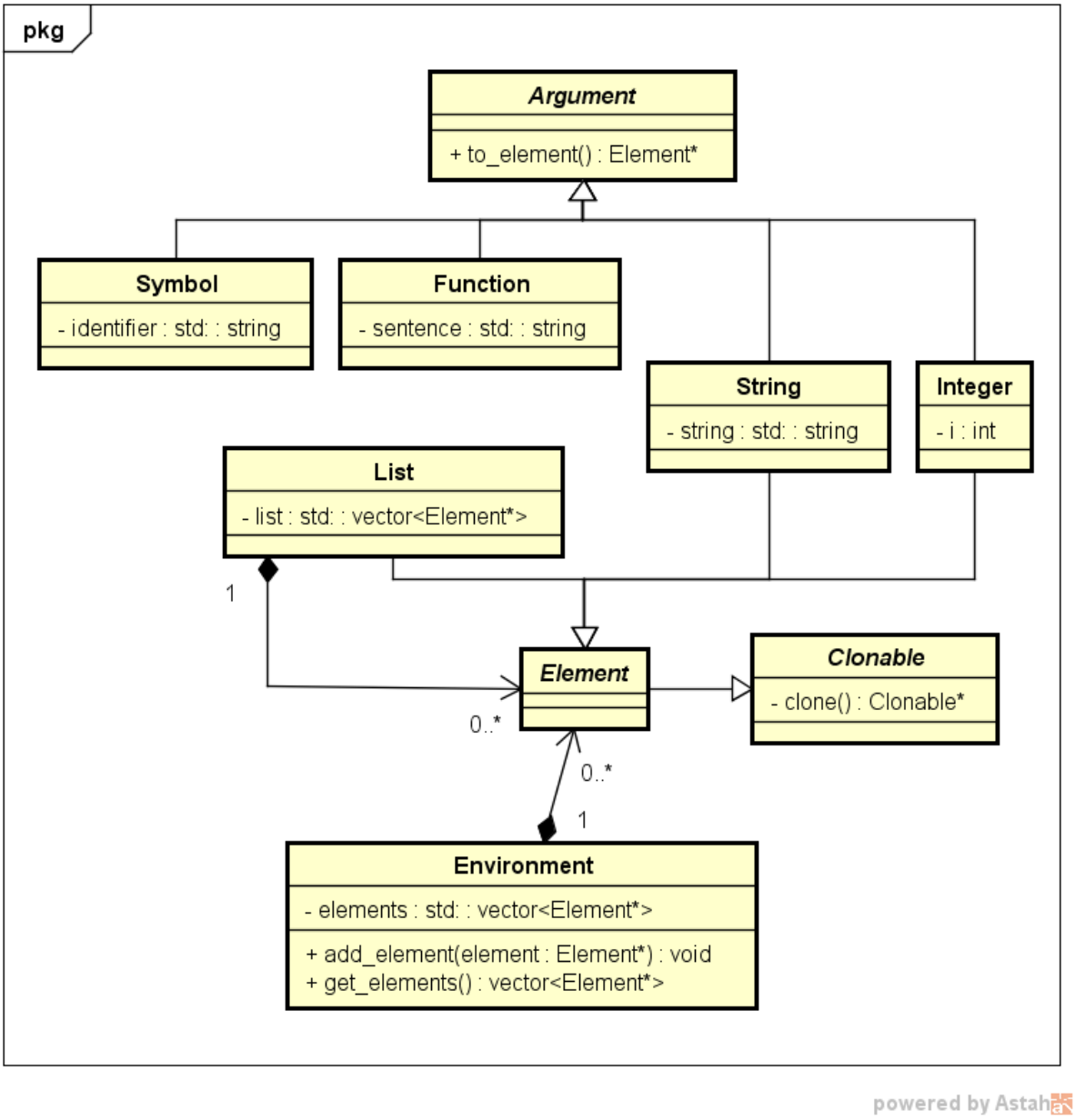

Figura 3: Environment

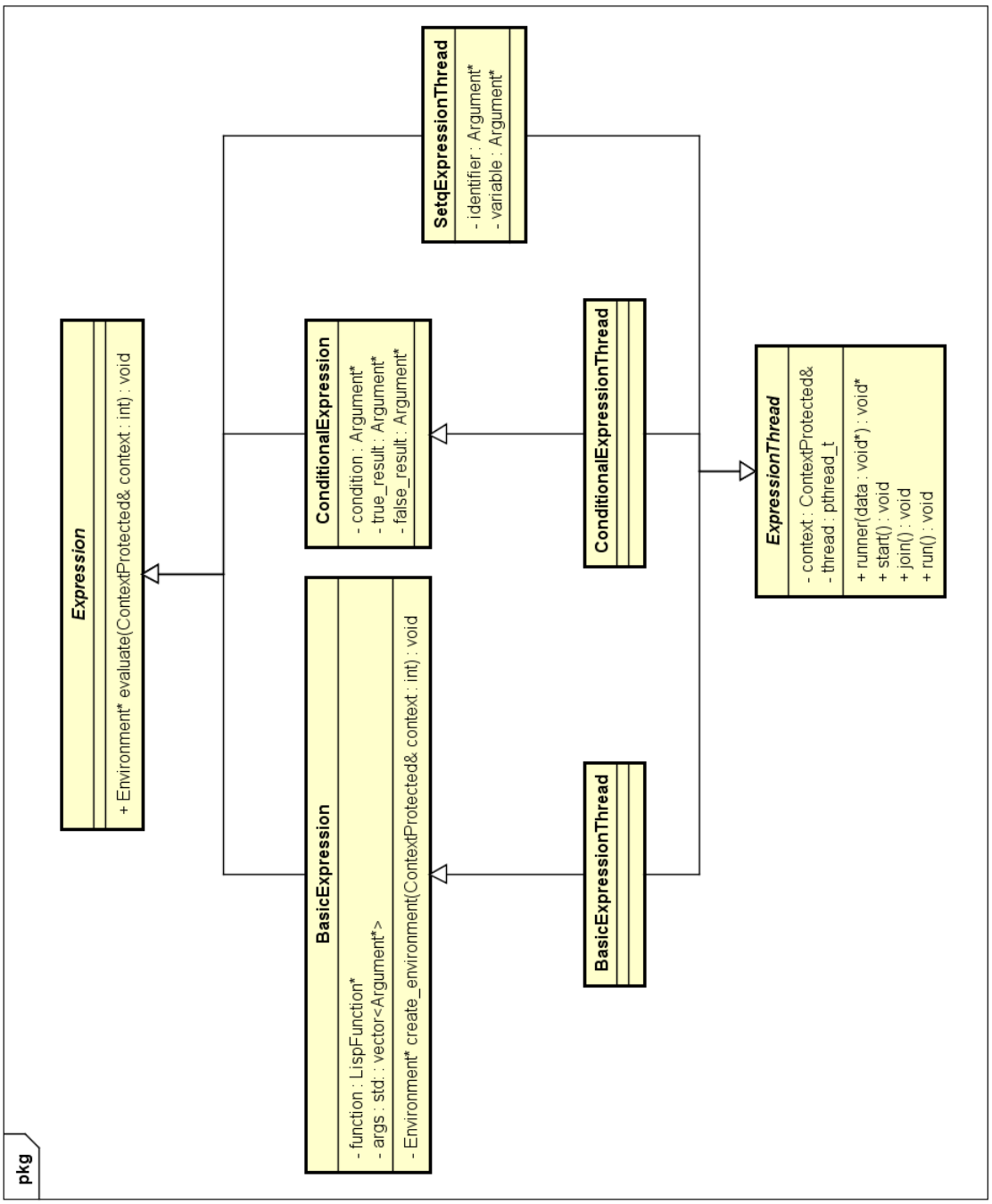

Figura 4: Expression

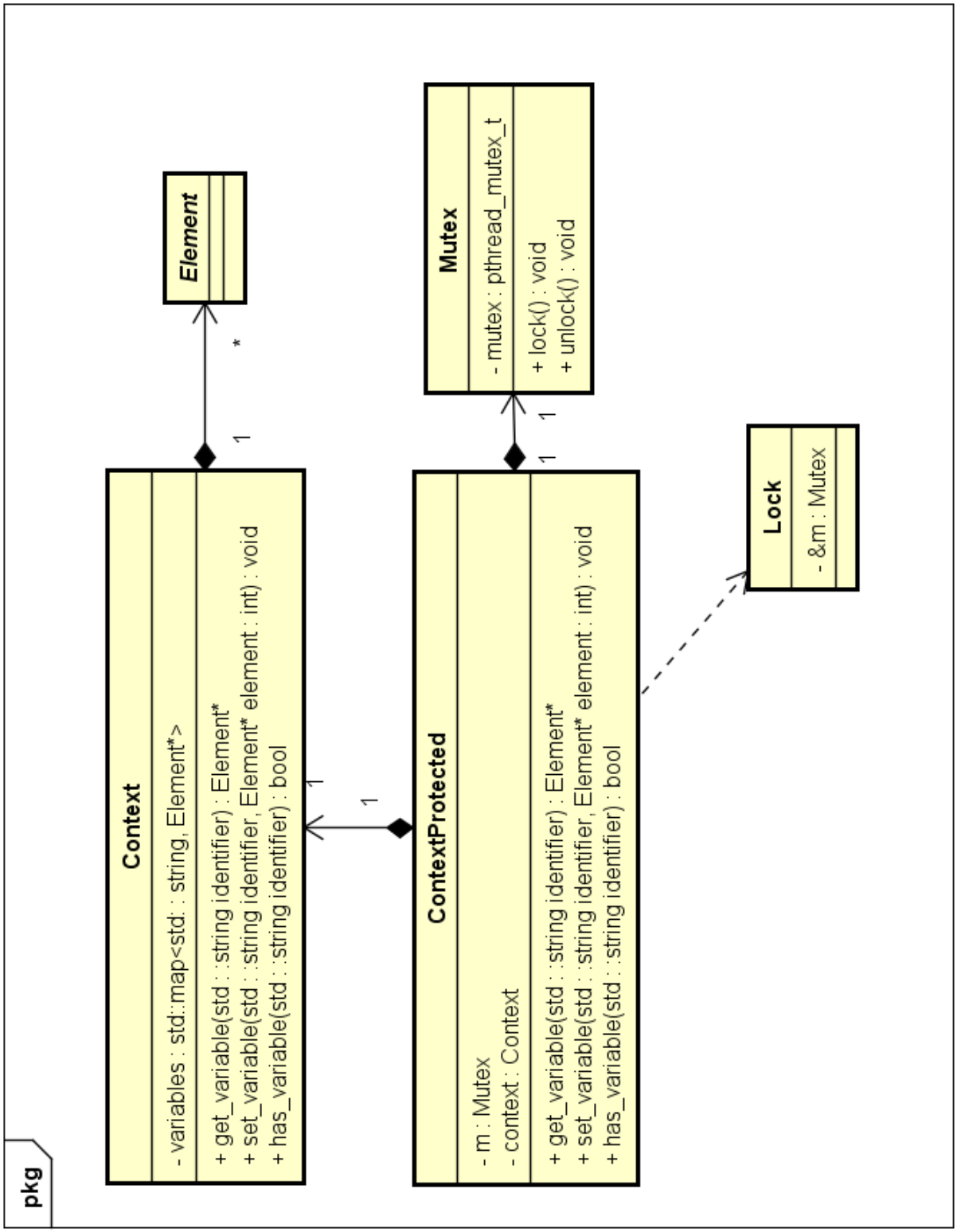

Figura 5: Context