μJava

Artur Ventura, Claudio Diniz, Miguel Nogueira

Implementação

Solução

```
> int foo = 2;
```

```
> int bar(){
return foo;
}
```

```
> bar()
```

```
class $Eval0 extends TopLevelEval{
public static int foo = 2;
}
```

```
class $Eval1 extends $Eval0{
 public static int bar(){
  return foo;
 }
}
```

```
class $Eval2 extends $Eval1{
 public static int $eval(){
  return (bar());
 }
}
```

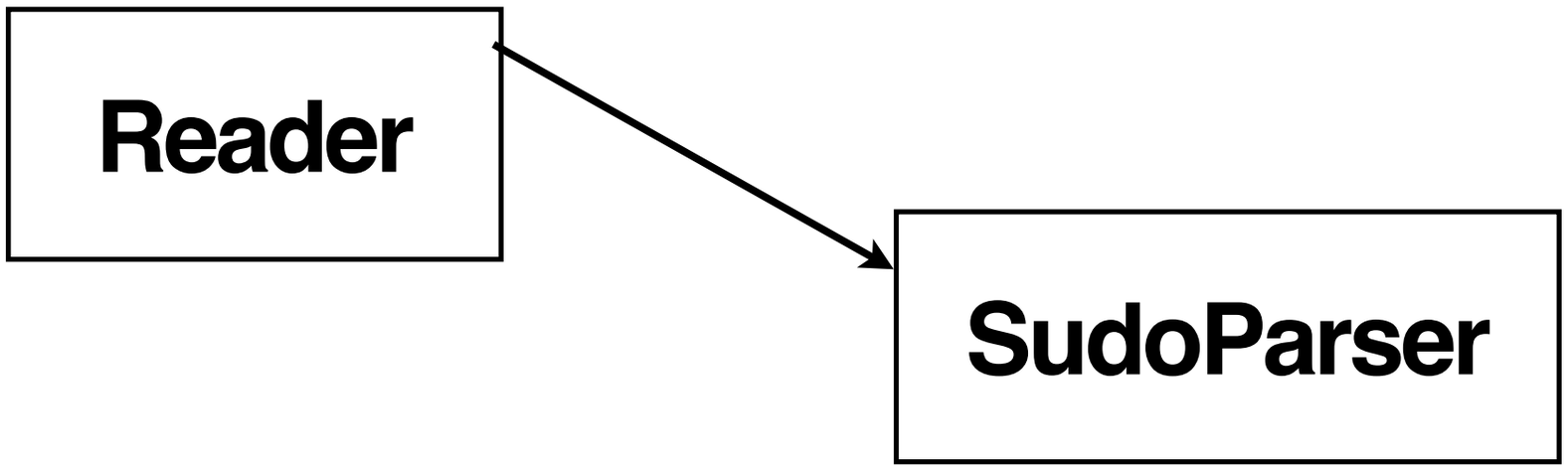
Implementação

Arquitectura da solução

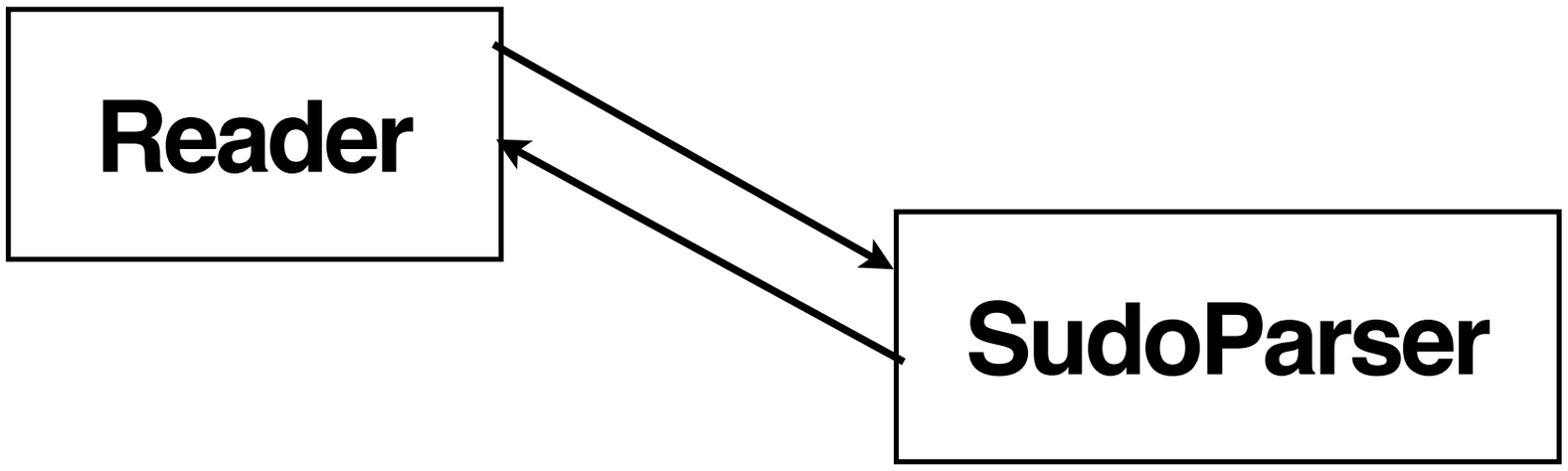
De forma a modularizar a criação de novas funcionalidades, implementamos o padrão de desenho Chain of Responsability, em que cada link da cadeia é responsável por uma funcionalidade.

Quando um link detecta uma expressão da sua responsabilidade, é executado e retorna uma expressão que tem associada uma Strategy (implementação do padrão Strategy) que sabe como resolver as expressões daquela forma

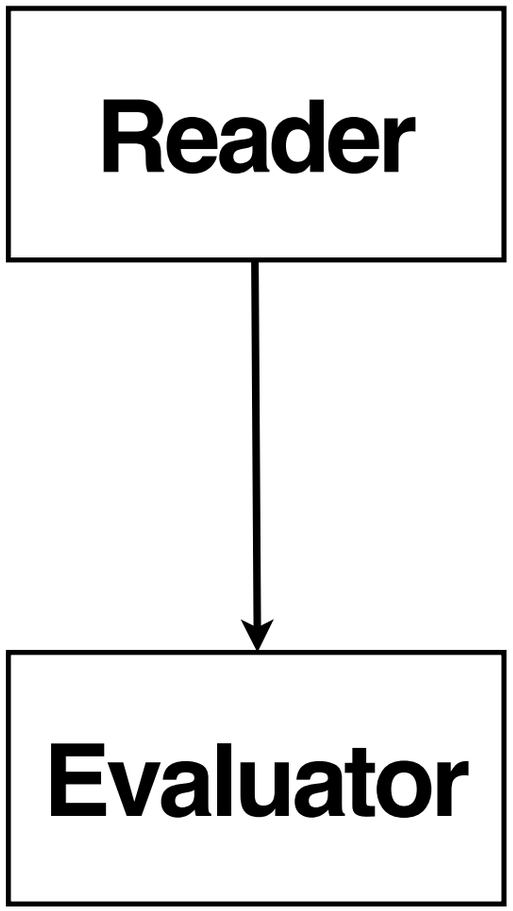
Reader



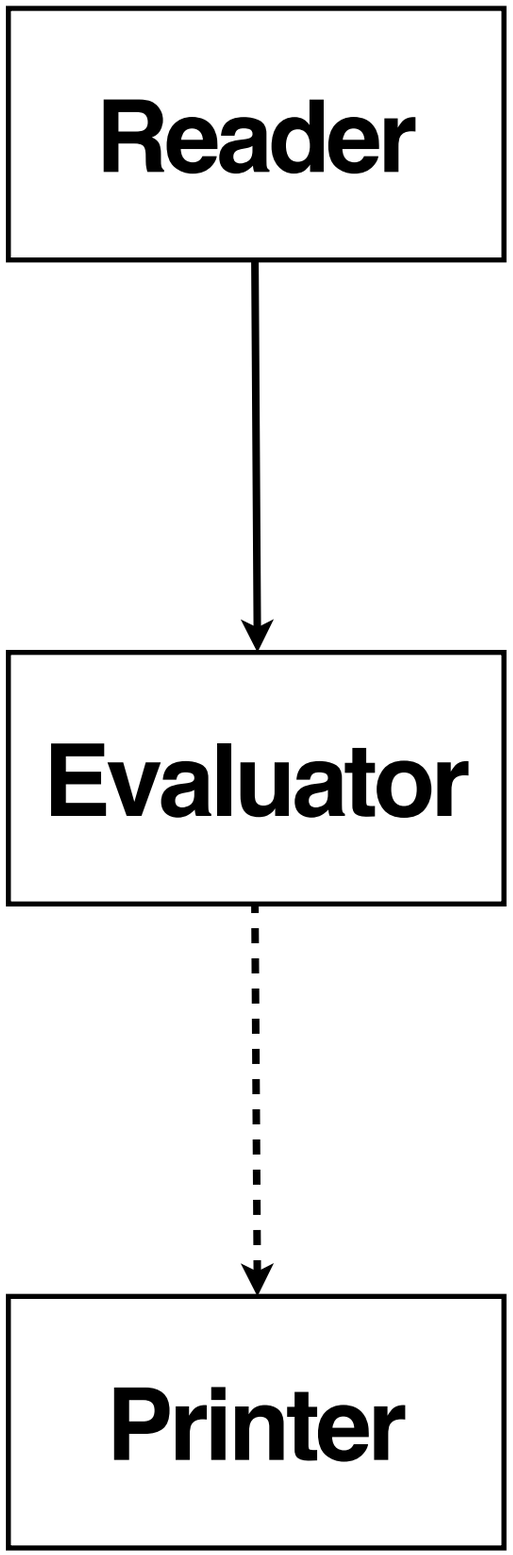
Nesta classe existe uma lista de ParsingLinks pelo qual vai ser filtrado o input, e caso seja da sua responsabilidade, é devolvida uma Expression do seu tipo



Nesta classe existe uma lista de ParsingLinks pelo qual vai ser filtrado o input, e caso seja da sua responsabilidade, é devolvida uma Expression do seu tipo



Nesta classe é invocado o metodo execute da Strategy da Expression. Caso a Strategy seja *evaluable* é invocado o metodo $eval sobre a classe que geramos.



Nesta classe é invocado o metodo execute da Strategy da Expression. Caso a Strategy seja *evaluable* é invocado o metodo $eval sobre a classe que geramos.

Se a Strategy da Expression implementa Printable então é invocado o metodo print sobre esta.

Implementação

Instruções

```
> bar();
```

```
class $EvalX extends $EvalX-1{
 public static void $eval(){
  bar();
 }
}
```

Implementação

Expressões

```
> 2 + 3
```

```
class $EvalX extends $EvalX-1{
 public static Object $eval(){
  return (InternalRepresentation.convert(2 + 3));
 }
}
```

`InternalRepresentation.convert` Devolve uma string com a representação do objecto em Java. Funciona como uma espécie de quote.

Implementação

Variaveis

```
> int foo = 2;
```

```
class $EvalX extends $EvalX-1{
  public static int foo = 2;
}
```

Implementação

Funções

```
> int bar(){
return foo;
}
```

```
class $EvalX extends $EvalX-1{
 public static int bar(){
  return foo;
 }
}
```

Implementação

Import

A string é passada para um Gestor de Imports. Quando é pedida uma ClassPool, são lhe adicionados os pacotes de import

Implementação

Static Import

```
> static import java.lang.Math;
```

```
class $EvalX extends $EvalX-1{
  public static double PI = java.lang.Math.PI;
  public static double sin(double arg0){
     return java.lang.Math.sin(arg0);
  }
  ...
}
```

A informação sobre a classe é capturada por reflexão.

Extensões

Suporte para expressões de fluxo de controlo:

```
> if (name == “Foo”){
  System.out.println(“bar”);
}
```

Foi apenas necessário criar um ParsingLink para detectar as expressões de fluxo de controlo e criar uma StatementExpression com esse conteúdo. A implementação é dada automaticamente pela StatementStrategy. São suportados `if`, `for`, `while`, `switch` e `do`;

Suporte para multi-line

Para permitir o multi-line foi necessário criar um grafo que analise o fluxo de entrada, e que consiga contar o numero de aberturas e fecho de chavetas, ignorando Strings e caracteres.

Extensões

Suporte para blocos nativos de java

```
> {
System.out.println(“hello”);
System.out.println(“world”);
}
```

Este caso é detectado pela StatementLink.

Suporte para excepções de Javassist

```
> foo()
[source error] foo() not found in $Eval0
> i
undeclared variable i
>
```

São detectadas algumas excepções de JavaAssist, relacionadas com problemas comuns (Erros de Sintaxe, Variáveis Não declaradas, etc.)

Extensões

Recursão Mutua & Funções dinamicas

De forma a ultrapassar a compilação estática entre as funções decidimos implementar o conceito de funções dinâmicas.

Estas são funções que a sua definição não depende do ambiente de compilação mas do ambiente de execução. De forma simples são funções em que é possível redefinir o seu corpo sem ser necessário recompilar todas as outras funções que a referenciam.

Extensões

Recursão Mutua & Funções dinamicas

```
> dynamic int foo(int i) {
    return 0;
}
> int bar (int z) {
    return foo(z);
}
> bar (4)
0
```

```
> dynamic int foo(int i) {
    return 1;
}
> bar (4)
1
```

Esta extensão é implementada através de uma tabela de indirecção que guarda a ultima definição da função.
O corpo de `foo` contem o código necessário á desreferenciação da tabela e da invação do metodo.

Extensões

Recursão Mutua & Funções dinamicas

Com esta solução consegue-se criar recursão mutua entre funções. No entanto existem a existência de funções estáticas e funções dinâmicas leva a situações onde não se obtêm o comportamento esperado

```
> dynamic int foo(int i){}
> int bar (int z) {
... referencia a foo
}
> dynamic int foo(int i) {
... referencia a bar
}
> int bar (int z) {
... referencia a foo
}
> bar(0)
// bar()
// foo()
// bar() mas queria qual?
```

Extensões

Recursão Mutua & Blocos

```
> block {
int foo(int i) { if (i == 0) return 0; else return
bar(i - 1); }
int bar(int i) { if (i == 0) return 0; else return
foo(i - 1); }
}
```

São avaliações que ocorrem em simultâneo. Apenas permitem a declaração de variáveis e funções. Este solução garante a implementação de recursão mutua.

As declarações em bloco vão ser todas colocadas na mesma classe de avaliação por isso não possível declarar variáveis com o mesmo nome ou funções com a mesma assinatura.

Extensões

Wrapping de funções

```
> wrappable int fact(int v){
    if (v == 0){
       return 1;
    }else{
       return v * fact(v - 1);
    }
}
```

A declaração de uma função como wrappable vai permitir que seja feita realizada algumas tarefas antes ou depois da execução desta. A ideia é semelhante á combinação de metodos do CLOS e da Programação orientada a Aspectos.

Esta não tão potente com os anteriores. Não é possível evitar a execução do método, e a única coisa que é possível adicionar são métodos a que vão ser passados os argumentos da função e o seu resultado.

Extensões

Wrapping de funções

```
%(before|after) <wrappable function> (add|del) <interceptor>
```

É possivel adicionar varios interceptors a uma funçao wrappable.
Agora torna-se trival implementar o trace de funções:

```
> void traceBefore(String f,Object[] v){
    System.out.println(f + "("+v[0]+")" + "=>");
}
> void traceAfter(String f,Object[] v, int result){
    System.out.println(f + "("+v[0]+")" + "<= " + result);
}
> %before fact(int) add traceBefore(java.lang.String,[Ljava.lang.Object;);
> %after fact(int) add traceAfter(java.lang.String,[Ljava.lang.Object;,int);
```

Já existe implementado na classe `ist.leic.pa.utils.TracingManager` **os métodos** `trace` **e** `untrace` **que fazem o mostrado em cima para uma função wrappable.**

Extensões

Wrapping de funções

```
> ist.leic.pa.utils.TracingManager.trace("fact(int)");
> fact(6)
//fact(6) =>
//  fact(5) =>
//    fact(4) =>
//      fact(3) =>
//        fact(2) =>
//          fact(1) =>
//            fact(0) =>
//            fact(0) <= 1
//          fact(1) <= 1
//        fact(2) <= 2
//      fact(3) <= 6
//    fact(4) <= 24
//  fact(5) <= 120
//fact(6) <= 720
720
>
```

Extensões

Definição de Classes

```
> class Example {
    int i = 0;
    int test(){
        i++;
        return i;
    }
}
> Example j = new Example();
```

È possivel definir novas classes. Estas são muito simplificadas. Não permitem keywords de acesso, nem construtores. As definições de métodos têm que ser semelhantes aos das funções. Não é possivel declarar elementos como `static`. Os fields e os métodos são sempre públicos. É possível o extender de outras classes (quer criadas no REPL, quer de sistema). Basicamente funcionam como Beans.

Extensões

Macros

```
> macro /([0-9]+)\s*\+/ { $1$ * }
> 2 + 3
6
```

A macro é uma operação de alteração a uma expressão que é dada como input. É dado como argumento uma expressão regular que irá fazer match contra expressões dadas como input do REPL, e um corpo em que é descrito o resultado da operação. O algoritmo é o seguinte:

1. As variáveis (ex. `$1$`) são substituídas pelos respectivos grupos do match da expressão regular.
2. O conteudo dos blocos na forma `%{ ... }` e `&{ ... }` é avaliado e o seu resultado é colocado no seu lugar.
    1.1. No caso de `&{ ... }` a expressão resultante deve ser uma string e será colocado o conteudo desta.
    1.2. No caso de `%{ ... }` é colocado a representação em java do resultado da expressão
3. Finalmente é retornado uma Expression que é o resultado de passar a o resultado da macro para o SudoParser. Isto vai fazer com que possa existir macros encadeadas e recursivas.

Extensões

Macros

```
> macro /entity ([^;]*);/ { block{
int $1$ = 0;
String &{ "get" + "$1$".substring(0, 1).toUpperCase() +
      "$1$".substring(1).toLowerCase() } (){
   return %{ "$1$" + " is :"} + $1$;
}
}}
> entity dog;
> getDog()
dog is: 0
> dog
0
```

Existem as funções `macros` e `deleteMacro` da classe `MacroManager` premitem fazer a gestão de macros no ambiente.

Extensões

Macros & Implementação

Quando é criada uma macro é criada um ParsingLink por Javassist que vai ser adicionado ao inicio da ParsingChain do SudoParser. Este link executar o algoritmo descrito atrás manipulando as strings de entrada e o corpo da macro. Este algorimo pode ainda chamar o avaliador para os corpos dos blocos de avaliação.

Estas macros não são tão potentes como as de Lisp, mas são ligeiramente mais potentes que as de C. A ideia principal é que estamos a gerar codigo, que gera codigo, que vai ser avaliado.

Questões?