XBA: Uma linguagem orientada a objetos com tipagem estrutural

Henrique Stagni
stagni@gmail.com

Orientador: Prof. Francisco Reverbel
reverbel@ime.usp.br

16/17 de novembro de 2011

Conteúdo

Sistema de tipo

- É *um* método formal que analisa *sintaticamente* um programa e prova que certos *estados indesejáveis* nunca serão atingidos durante a execução.
- Exemplos de estados indesejáveis que sistemas de tipos tentam evitar incluem:
  - Soma de um inteiro com um caracter.
  - Uso de um valor não booleano como condição de um **if**.
  - Aplicação de função sobre algo que não é função.
  - etc...

Sistema de tipo

- É *um* método formal que analisa *sintaticamente* um programa e prova que certos *estados indesejáveis* nunca serão atingidos durante a execução.
- Exemplos de estados indesejáveis que sistemas de tipos tentam evitar incluem:
  - Soma de um inteiro com um caracter.
  - Uso de um valor não booleano como condição de um **if**.
  - Aplicação de função sobre algo que não é função.
  - etc...

Tipos e Subtipos

- Sistemas de tipos evitam estados indesejáveis, por meio da *classificação* de seus *termos*[1] de acordo com *características dos valores para os quais esses termos são computados*(**tipos**)
- De maneira informal, um tipo τ_1 é **subtipo** de τ_2 se é possível usar termos do tipo τ_1 em todas as construções que esperam por termos do tipo τ_2, sem afetar a validade do programa.

[1] qualquer construção sintática que, em tempo de execução, pode ser avaliada para um valor

Tipagem "estática" nominal

- Um tipo é identificado apenas pelo nome que é dado a ele durante a sua definição.
- A relação de subtipagem é fornecida *explicitamente* pelo programador.
- São os casos de linguagens como C, Java, etc.

Tipagem "estática" nominal

- Um tipo é identificado apenas pelo nome que é dado a ele durante a sua definição.
- A relação de subtipagem é fornecida *explicitamente* pelo programador.
- São os casos de linguagens como C, Java, etc.

Tipagem "estática" nominal

- Um tipo é identificado apenas pelo nome que é dado a ele durante a sua definição.
- A relação de subtipagem é fornecida *explicitamente* pelo programador.
- São os casos de linguagens como C, Java, etc.

Exemplo (pseudo-Java)


```
class Circle {
    int radius() { ... }
}

class ColorfulCircle {
    int radius() { ... }
    int color() { ... }
}

class Square {
    int side() { ... }
}

int diameter(Circle c) {
    return 2*c.radius();
}

diametro(circle);          // OK
diametro(colorfulCircle); // Erro em tempo de compilacao
diametro(square);          // Erro em tempo de compilacao.
```

Tipagem "dinâmica"

- Expressão utilizada para Linguagens, à rigor, que não possuem um sistema de tipos.
- São linguagens como *Smalltalk*, *Ruby*, *Python*.

Tipagem "dinâmica"

- Expressão utilizada para Linguagens, à rigor, que não possuem um sistema de tipos.
- São linguagens como *Smalltalk*, *Ruby*, *Python*.

Exemplo (pseudo-Ruby)


```
class Circle
    def radius ... end
end

class ColorfulCircle
    def radius ... end
    def color ... end
end

class Square
    def side ... end
end

def diameter(c)
    2*c.radius
end

diameter(circle)            # OK
diameter(colorfulCircle)    # OK
diameter(square)            # OK, erro durante execucao
```

Tipagem estrutural

- A estrutura dos tipos é utilizada para identificá-los e para estabelecer automaticamente as relações de subtipagem.

Exemplo (pseudo-XBA)


```
class Circle
    method radius! = ...

class ColorfulCircle
    method radius! = ...
    method color! = ...

class Square
    method side! = ...

method diameter: c =
    2*c.raio!

diameter: circle           # OK
diameter: colorfulCircle   # OK
diameter: square           # Erro em tempo de compilacao
```

Vantagens/Desvantagens

- Permitem uma maior flexibilidade do que a tipagem nominativa
- São capazes de detectar, em *tempo de compilação*, erros que uma linguagem dinâmica só acusaria em *tempo de execução*.
- Descrever anotações de tipo explicitamente é inviável – é necessário que haja seja feita uma inferência de tipos.
- Classes compatíveis estruturalmente podem não ser compatíveis semanticamente.

Vantagens/Desvantagens

- Permitem uma maior flexibilidade do que a tipagem nominativa
- São capazes de detectar, em *tempo de compilação*, erros que uma linguagem dinâmica só acusaria em *tempo de execução*.
- Descrever anotações de tipo explicitamente é inviável – é necessário que haja seja feita uma inferência de tipos.
- Classes compatíveis estruturalmente podem não ser compatíveis semanticamente.

Vantagens/Desvantagens

- Permitem uma maior flexibilidade do que a tipagem nominativa
- São capazes de detectar, em *tempo de compilação*, erros que uma linguagem dinâmica só acusaria em *tempo de execução*.
- Descrever anotações de tipo explicitamente é inviável – é necessário que haja seja feita uma inferência de tipos.
- Classes compatíveis estruturalmente podem não ser compatíveis semanticamente.

Vantagens/Desvantagens

- Permitem uma maior flexibilidade do que a tipagem nominativa
- São capazes de detectar, em *tempo de compilação*, erros que uma linguagem dinâmica só acusaria em *tempo de execução*.
- Descrever anotações de tipo explicitamente é inviável – é necessário que haja seja feita uma inferência de tipos.
- Classes compatíveis estruturalmente podem não ser compatíveis semanticamente.

Conteúdo

Proposta do trabalho

Consistiu em:

- Especificar uma linguagem de programação com suporte à orientação a objetos e com tipagem estrutural
- Desenvolver um compilador para ela.

O *objetivo* principal desse trabalho consistiu no estudo de sistemas de tipos estruturais e de métodos para se realizar inferência de tipos em sistemas como esse.

Proposta do trabalho

Consistiu em:

- Especificar uma linguagem de programação com suporte à orientação a objetos e com tipagem estrutural
- Desenvolver um compilador para ela.

O *objetivo* principal desse trabalho consistiu no estudo de sistemas de tipos estruturais e de métodos para se realizar inferência de tipos em sistemas como esse.

Proposta do trabalho

Consistiu em:

- Especificar uma linguagem de programação com suporte à orientação a objetos e com tipagem estrutural
- Desenvolver um compilador para ela.

O *objetivo* principal desse trabalho consistiu no estudo de sistemas de tipos estruturais e de métodos para se realizar inferência de tipos em sistemas como esse.

Características

- Sistema de tipos estrutural.
- Suporte *parcial* à orientação a objetos.
- Funções como objetos de primeira classe.
- Todos os métodos/funções usam o conceito de *currying*– podemos considerar que todos os métodos recebem exatamente 1 parâmetro.
- Outras características/detalhes menos importantes tais como:
  - Variáveis (*boxes*) e todos os literais são objetos.
  - Uso de seletores para chamadas (*Smalltalk*) de método
  - Uso da indentação para definir blocos de expressões (*Python*).

Características

- Sistema de tipos estrutural.
- Suporte *parcial* à orientação a objetos.
- Funções como objetos de primeira classe.
- Todos os métodos/funções usam o conceito de *currying*– podemos considerar que todos os métodos recebem exatamente 1 parâmetro.
- Outras características/detalhes menos importantes tais como:
  - Variáveis (*boxes*) e todos os literais são objetos.
  - Uso de seletores para chamadas (*Smalltalk*) de método
  - Uso da indentação para definir blocos de expressões (*Python*).

Características

- Sistema de tipos estrutural.
- Suporte *parcial* à orientação a objetos.
- Funções como objetos de primeira classe.
- Todos os métodos/funções usam o conceito de *currying*– podemos considerar que todos os métodos recebem exatamente 1 parâmetro.
- Outras características/detalhes menos importantes tais como:
  - Variáveis (*boxes*) e todos os literais são objetos.
  - Uso de seletores para chamadas (*Smalltalk*) de método
  - Uso da indentação para definir blocos de expressões (*Python*).

Características

- Sistema de tipos estrutural.
- Suporte *parcial* à orientação a objetos.
- Funções como objetos de primeira classe.
- Todos os métodos/funções usam o conceito de *currying*– podemos considerar que todos os métodos recebem exatamente 1 parâmetro.
- Outras características/detalhes menos importantes tais como:
  - Variáveis (*boxes*) e todos os literais são objetos.
  - Uso de seletores para chamadas (*Smalltalk*) de método
  - Uso da indentação para definir blocos de expressões (*Python*).

Características

- Sistema de tipos estrutural.
- Suporte *parcial* à orientação a objetos.
- Funções como objetos de primeira classe.
- Todos os métodos/funções usam o conceito de *currying*– podemos considerar que todos os métodos recebem exatamente 1 parâmetro.
- Outras características/detalhes menos importantes tais como:
  - Variáveis (*boxes*) e todos os literais são objetos.
  - Uso de seletores para chamadas (*Smalltalk*) de método
  - Uso da indentação para definir blocos de expressões (*Python*).

Exemplo


```
class Circle with-radius: radius
  field radius = radius
  method radius! = radius #devolve o valor do campo radius

class ColoredCircle with-radius: radius and-color:color
  field radius = radius
  field color = color

  method radius! = radius
  method color! = color

class Main
  method print-diamenter: circle =
    print (2*circle radius!)

  method init!  #Ponto de entrada do programa =
    let circle = Circle new-with-radius:26,
        colored = Circle new-with-radius:4 and-color:0 in
      self print-diameter: circle   # OK
      self print-diameter: colored  # OK
      self print-diamenter: 88      # Sistema de tipos acusa
```

O sistema de tipos

- Tem o objetivo de evitar que se faça chamadas a um método em um objeto que não o possui – esses são os estados indesejáveis do sistema de tipos.
- Baseado no sistema *Hindley-Milner* de tipos, que é a base para o sistema de tipos de diversas linguagens:
  - Haskell
  - Clean
  - ML, Ocaml, F#
- Usada extensão que acrescenta *records* ao sistema de tipos.

O sistema de tipos

- Tem o objetivo de evitar que se faça chamadas a um método em um objeto que não o possui – esses são os estados indesejáveis do sistema de tipos.
- Baseado no sistema *Hindley-Milner* de tipos, que é a base para o sistema de tipos de diversas linguagens:
  - Haskell
  - Clean
  - ML, Ocaml, F#
- Usada extensão que acrescenta *records* ao sistema de tipos.

O sistema de tipos

- Tem o objetivo de evitar que se faça chamadas a um método em um objeto que não o possui – esses são os estados indesejáveis do sistema de tipos.
- Baseado no sistema *Hindley-Milner* de tipos, que é a base para o sistema de tipos de diversas linguagens:
  - Haskell
  - Clean
  - ML, Ocaml, F#
- Usada extensão que acrescenta *records* ao sistema de tipos.

Os tipos

- Tipos descrevem a estrutura de um objeto.
- Tipos são formados por triplas (uma por método) da forma (τ_1, L, τ_2), onde:
  - L é um label que representa o *seletor* do método associado.
  - τ_1 é o tipo do valor devolvido pelo método associado.
  - τ_2 é o tipo do parâmetro do método associado.
- adicionalmente, tipos podem ser *fechados* ou *abertos*. tipos *fechados*/*abertos* estão associados a objetos que possuem *exatamente*/*pelo menos* aquele conjunto de métodos.

Os tipos

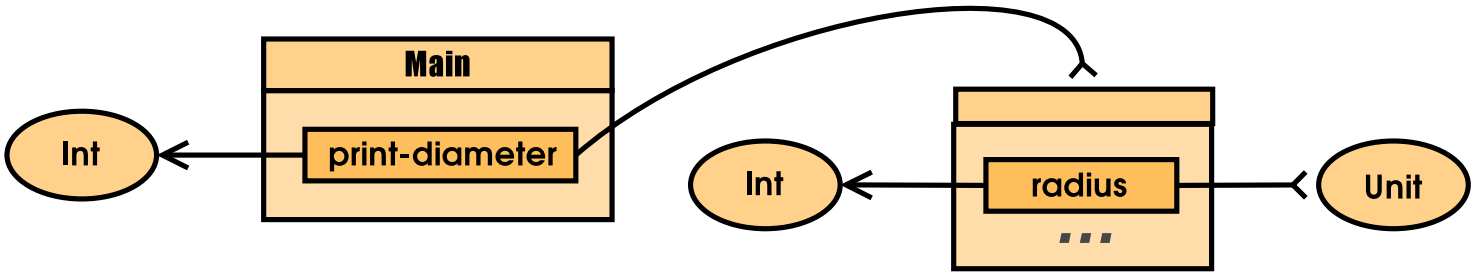
- Tipos descrevem a estrutura de um objeto.
- Tipos são formados por triplas (uma por método) da forma (τ_1, L, τ_2), onde:
  - L é um label que representa o *seletor* do método associado.
  - τ_1 é o tipo do valor devolvido pelo método associado.
  - τ_2 é o tipo do parâmetro do método associado.
- adicionalmente, tipos podem ser *fechados* ou *abertos*. tipos *fechados*/*abertos* estão associados a objetos que possuem *exatamente*/*pelo menos* aquele conjunto de métodos.

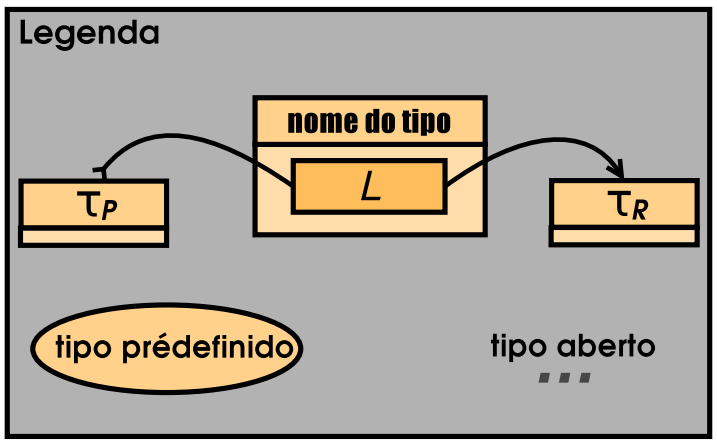
Os tipos

- Tipos descrevem a estrutura de um objeto.
- Tipos são formados por triplas (uma por método) da forma (τ_1, L, τ_2), onde:
  - L é um label que representa o *seletor* do método associado.
  - τ_1 é o tipo do valor devolvido pelo método associado.
  - τ_2 é o tipo do parâmetro do método associado.
- adicionalmente, tipos podem ser *fechados* ou *abertos*. tipos *fechados*/*abertos* estão associados a objetos que possuem *exatamente*/*pelo menos* aquele conjunto de métodos.

Exemplo

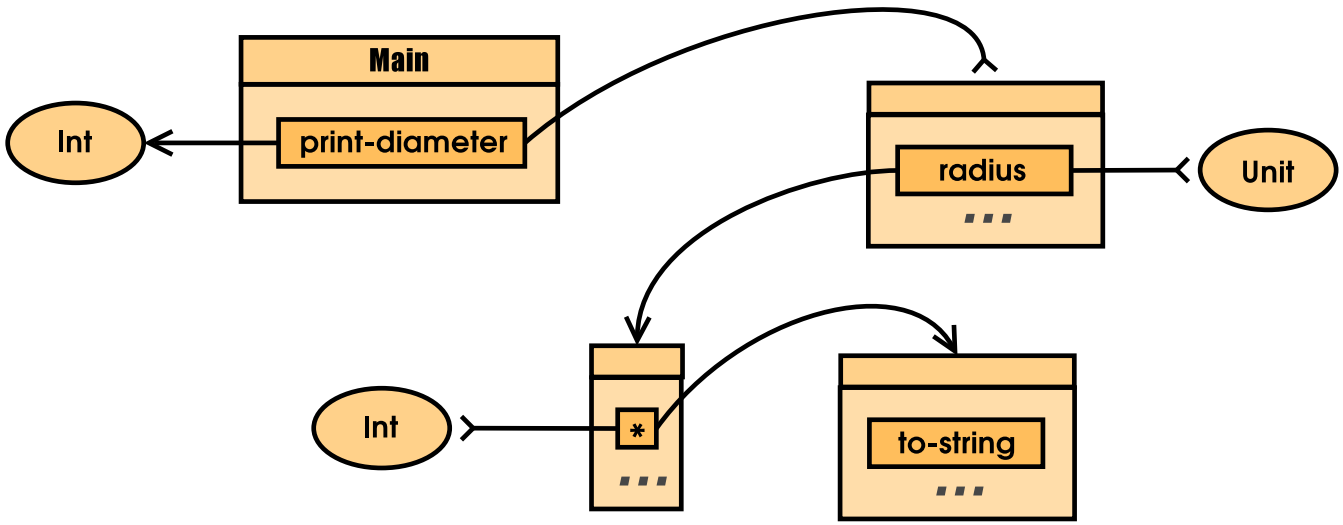
```
class Main
    method print-diameter: c =
        print (2 * c radius!)
```





Exemplo

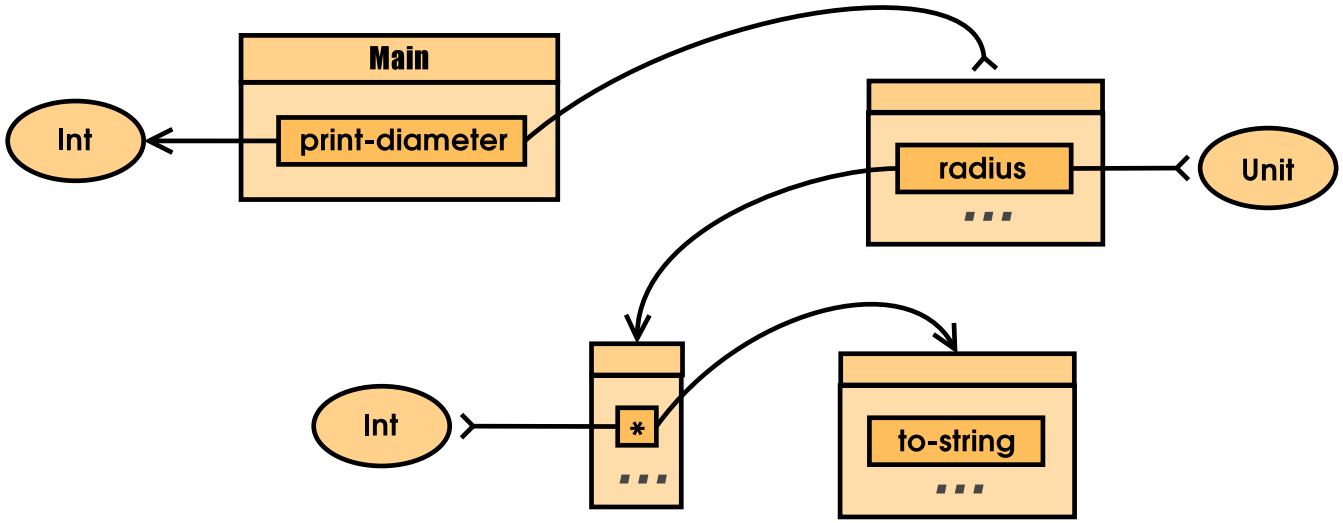

```
class Main
    method print-diameter: c =
        print (c radius! * 2)
```



Tipos podem se tornar complexos rapidamente, mesmo em exemplos simples. Por isso é necessário que exista um sistema de **inferência** de tipos

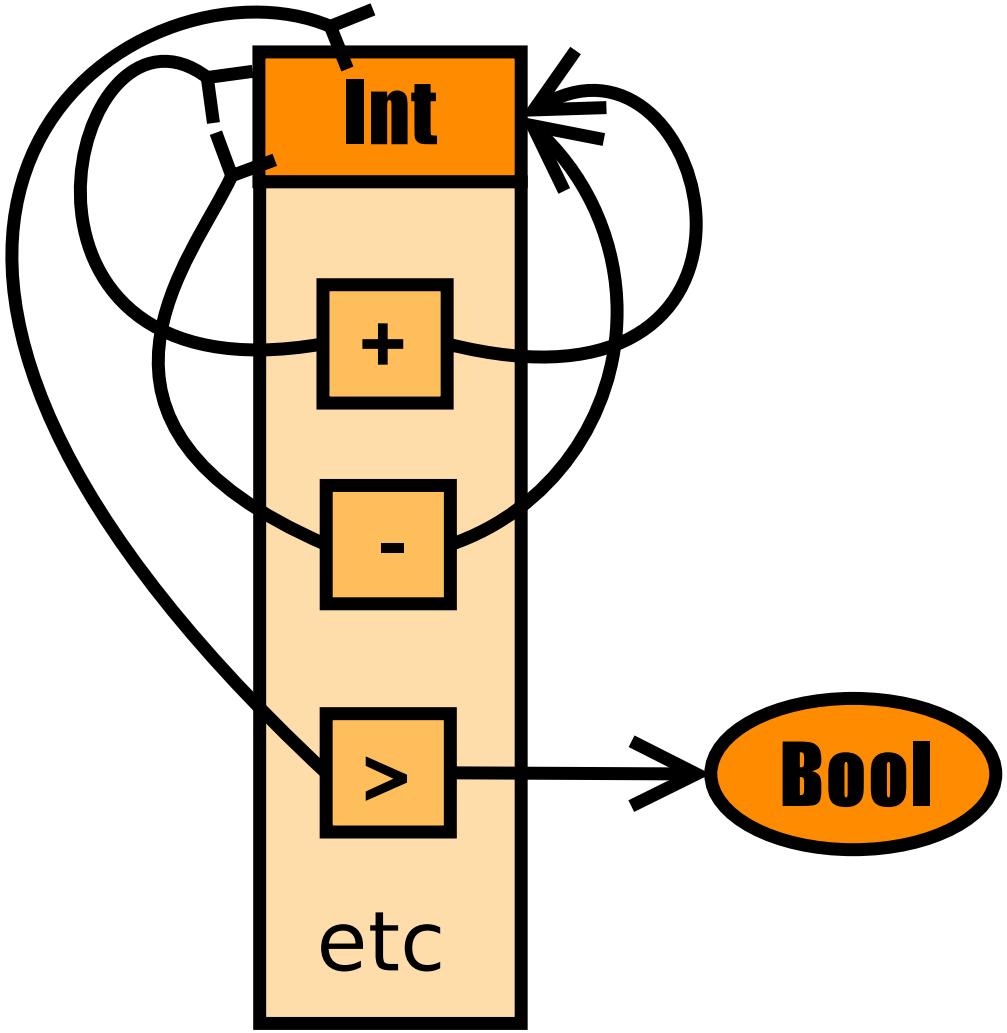
Exemplo


```
class Main
    method print-diameter: c =
        print (c radius! * 2)
```



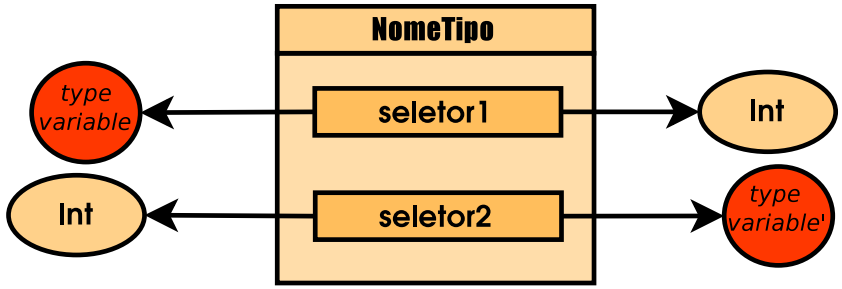
Tipos podem se tornar complexos rapidamente, mesmo em exemplos simples. Por isso é necessário que exista um sistema de **inferência** de tipos

Exemplo



Unificação

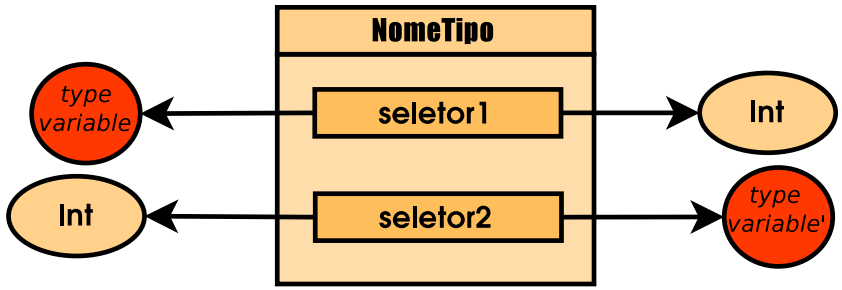
- Em lógica, unificação é o termo usado para descrever o processo de encontrar, quando possível, uma *substituição* de variáveis por fórmulas, com o intuito de tornar duas fórmulas idênticas; esse processo será aplicado **dentro do contexto dos tipos** da linguagem desenvolvida.



- Para isso, estendemos nossa definição de tipos de tal forma que um tipo também possa ser uma *type variable* – uma variável que representa um tipo não determinado.

Unificação

- Em lógica, unificação é o termo usado para descrever o processo de encontrar, quando possível, uma *substituição* de variáveis por fórmulas, com o intuito de tornar duas fórmulas idênticas; esse processo será aplicado **dentro do contexto dos tipos** da linguagem desenvolvida.



- Para isso, estendemos nossa definição de tipos de tal forma que um tipo também possa ser uma *type variable* – uma variável que representa um tipo não determinado.

Unificação

- Dados dois tipos τ_1 e τ_2, o processo de unificação consiste em encontrar, *se possível* um conjunto de substituições de *type variables* por tipos que torne τ_1 e τ_2 idênticos.
- Nesse contexto, um tipo *aberto* é visto como se contivesse infinitas triplas da forma (τ_P, L, τ_R), para todo L diferente das labels das triplas iniciais (com τ_P e τ_R sendo *type-variables*) – isso é uma simplificação do conceito de *row variables* de sistemas de tipos com *records*.
  - **Na prática, isso quer dizer que a unificação pode 'adicionar' triplas a tipos abertos**

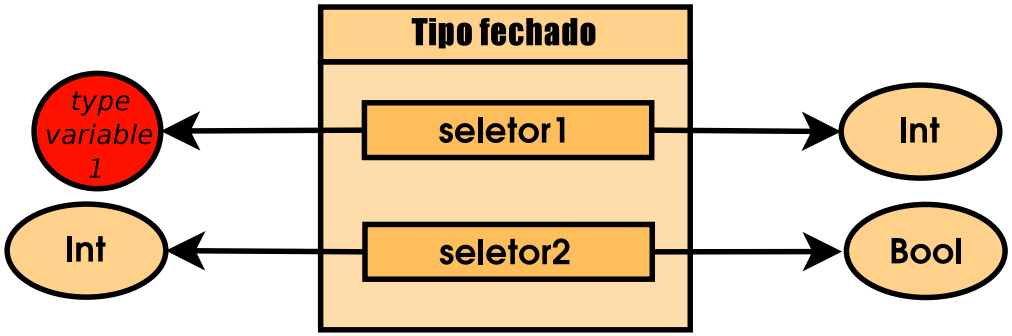
Unificação

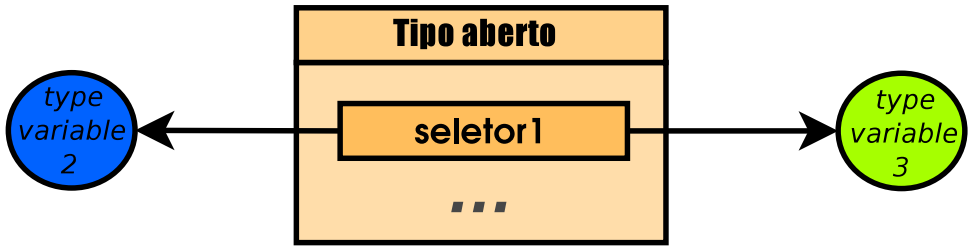
- Dados dois tipos τ_1 e τ_2, o processo de unificação consiste em encontrar, *se possível* um conjunto de substituições de *type variables* por tipos que torne τ_1 e τ_2 idênticos.
- Nesse contexto, um tipo *aberto* é visto como se contivesse infinitas triplas da forma (τ_P, L, τ_R), para todo L diferente das labels das triplas iniciais (com τ_P e τ_R sendo *type-variables*) – isso é uma simplificação do conceito de *row variables* de sistemas de tipos com *records*.
    - Na prática, isso quer dizer que a unificação pode 'adicionar' triplas a tipos abertos

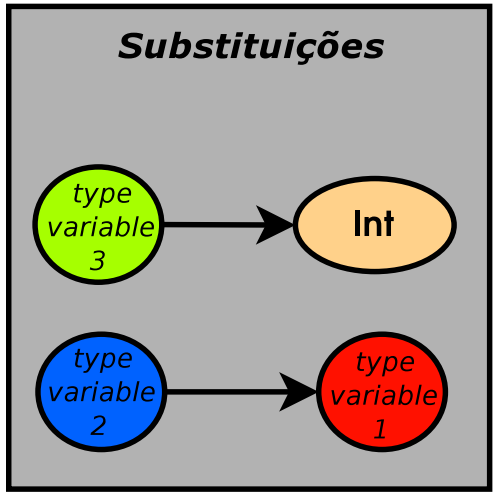
Unificação

- Dados dois tipos τ_1 e τ_2, o processo de unificação consiste em encontrar, *se possível* um conjunto de substituições de *type variables* por tipos que torne τ_1 e τ_2 idênticos.
- Nesse contexto, um tipo *aberto* é visto como se contivesse infinitas triplas da forma (τ_P, L, τ_R), para todo L diferente das labels das triplas iniciais (com τ_P e τ_R sendo *type-variables*) – isso é uma simplificação do conceito de *row variables* de sistemas de tipos com *records*.
  - **Na prática, isso quer dizer que a unificação pode 'adicionar' triplas a tipos abertos**

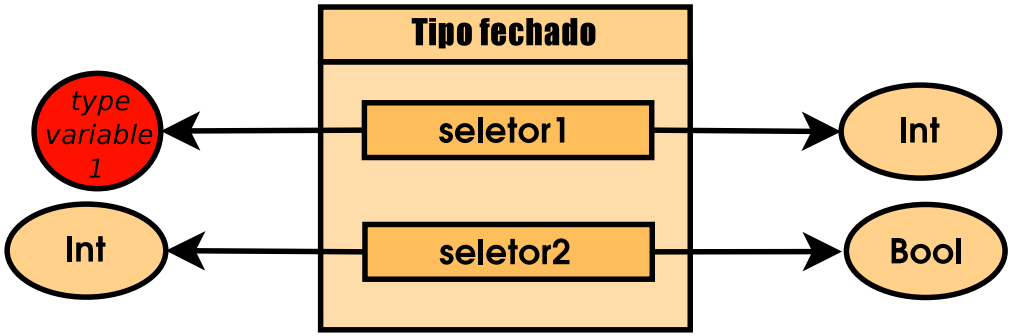
Unificação - Exemplos





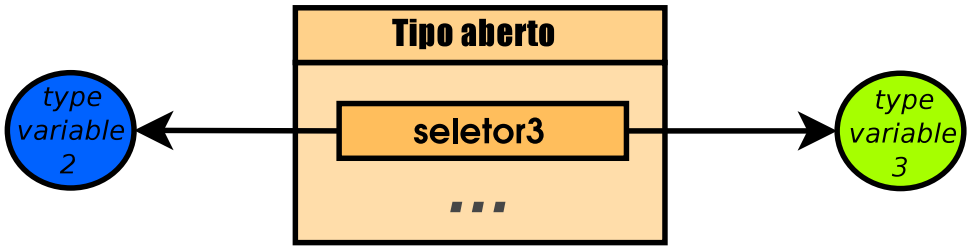


Unificação - Exemplos



Substituições

Não existe substituição que unifique esses tipos



Unificação - algoritmo

- Recebe como entrada dois tipos τ_1, τ_2; devolve **SIM** se é possível e unificá-los e **NÃO** caso contrário.
- $\overline{\tau_1} := \text{find}(\tau_1), \overline{\tau_2} := \text{find}(\tau_2)$.
- Se $\overline{\tau_1} = \overline{\tau_2}$ devolve **SIM**.
- Se $\overline{\tau_1}$ é uma *type variable*, $\text{union}(\overline{\tau_1}, \overline{\tau_2})$. Devolve **SIM**. (mesma coisa para $\overline{\tau_2}$...)
- Se os tipos possuem o mesmo conjunto de labels (ou se é possível estender tipos abertos para tal), $\text{union}(\overline{\tau_1}, \overline{\tau_2})$. Devolve **conjunção das unificações** dos τ_R e τ_P das triplas correspondentes.
- Caso contrário, devolve **NÃO**

union e find usuais, mas tomando certos cuidados como não deixar que uma type variable seja a representante de um conjunto.

Unificação - algoritmo

- Recebe como entrada dois tipos τ_1, τ_2; devolve **SIM** se é possível e unificá-los e **NÃO** caso contrário.
- $\overline{\tau_1} := \text{find}(\tau_1), \overline{\tau_2} := \text{find}(\tau_2)$.
- Se $\overline{\tau_1} = \overline{\tau_2}$ devolve **SIM**.
- Se $\overline{\tau_1}$ é uma *type variable*, $\text{union}(\overline{\tau_1}, \overline{\tau_2})$. Devolve **SIM**. (mesma coisa para $\overline{\tau_2}$...)
- Se os tipos possuem o mesmo conjunto de labels (ou se é possível estender tipos abertos para tal), $\text{union}(\overline{\tau_1}, \overline{\tau_2})$. Devolve **conjunção das unificações** dos τ_R e τ_P das triplas correspondentes.
- Caso contrário, devolve **NÃO**

union e find usuais, mas tomando certos cuidados como não deixar que uma type variable seja a representante de um conjunto.

Unificação - Detalhes adicionais

- Se a unificação for bem sucedida, basta fazer find(τ_v) para saber por que tipo a variável τ_v deve ser substituída na unificação.
- Essa implementação permite que a unificação utilize substituições circulares (ou recursivas), o que é importante para que tipos recursivos sejam inferidos.

Unificação - Detalhes adicionais

- Se a unificação for bem sucedida, basta fazer find(τ_v) para saber por que tipo a variável τ_v deve ser substituída na unificação.
- Essa implementação permite que a unificação utilize substituições circulares (ou recursivas), o que é importante para que tipos recursivos sejam inferidos.

Inferência de tipos

Para cada expressão da linguagem:

- Infere-se o tipo de suas *subexpressões*.
- Constroi-se o tipos esperados das subexpressões de forma que a expressão inicial seja válida. Esses tipos geralmente são construidos usando *type variables*.
- Unifica-se os tipos inferidos com os tipos esperados das subexpressões.
- Extrai-se o tipo da expressão (fazendo as substituições decorrentes das unificações)

Inferência de tipos

Para cada expressão da linguagem:

- Infere-se o tipo de suas *subexpressões*.
- Constroi-se o tipos esperados das subexpressões de forma que a expressão inicial seja válida. Esses tipos geralmente são construidos usando *type variables*.
- Unifica-se os tipos inferidos com os tipos esperados das subexpressões.
- Extrai-se o tipo da expressão (fazendo as substituições decorrentes das unificações)

Inferência de tipos

Para cada expressão da linguagem:

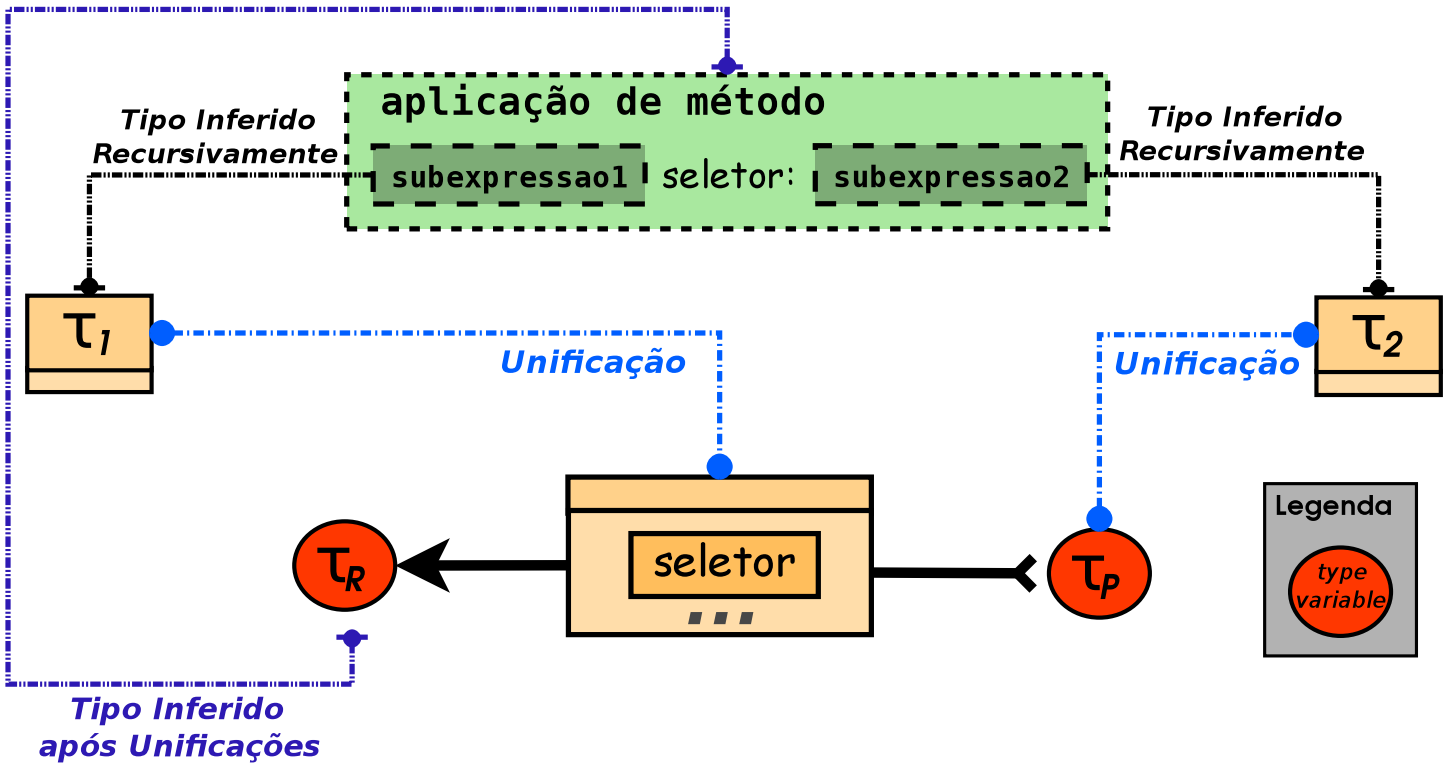
- Infere-se o tipo de suas *subexpressões*.
- Constroi-se o tipos esperados das subexpressões de forma que a expressão inicial seja válida. Esses tipos geralmente são construidos usando *type variables*.
- Unifica-se os tipos inferidos com os tipos esperados das subexpressões.
- Extrai-se o tipo da expressão (fazendo as substituições decorrentes das unificações)

Inferência de tipos

Para cada expressão da linguagem:

- Infere-se o tipo de suas *subexpressões*.
- Constroi-se o tipos esperados das subexpressões de forma que a expressão inicial seja válida. Esses tipos geralmente são construidos usando *type variables*.
- Unifica-se os tipos inferidos com os tipos esperados das subexpressões.
- Extrai-se o tipo da expressão (fazendo as substituições decorrentes das unificações)

Inferência de tipos - Exemplo



Detalhes adicionais

- Se em qualquer momento, a unificação devolver NÃO, então o programa não é aceito pelo sistema de tipos.
- Os tipos extraídos podem conter *variáveis livres* (na implementação, são variáveis τ_v tais que $\text{find}(\tau_v) = \tau_v$). É algo que pode acontecer naturalmente em funções genéricas como *identidade*, *map*, etc.

Detalhes adicionais

- Se em qualquer momento, a unificação devolver NÃO, então o programa não é aceito pelo sistema de tipos.
- Os tipos extraídos podem conter *variáveis livres* (na implementação, são variáveis τ_v tais que $\text{find}(\tau_v) = \tau_v$). É algo que pode acontecer naturalmente em funções genéricas como *identidade*, *map*, etc.

Conteúdo

Porque a escolha?

- Os próprios bytecodes possuem nível de abstração alto – mais fácil gerar código.
- Conceito de objetos (métodos, construtores, herança, etc) já está presente no nível dos bytecodes.
- **Não é necessário se preocupar muito com otimização na geração dos bytecodes.**. Otimizações agressivas são feitas pelo JIT.
- Multiplaforma, imensa coleção de bibliotecas, etc.

Porque a escolha?

- Os próprios bytecodes possuem nível de abstração alto – mais fácil gerar código.
- Conceito de objetos (métodos, construtores, herança, etc) já está presente no nível dos bytecodes.
- **Não é necessário se preocupar muito com otimização na geração dos bytecodes.**. Otimizações agressivas são feitas pelo JIT.
- Multiplaforma, imensa coleção de bibliotecas, etc.

Porque a escolha?

- Os próprios bytecodes possuem nível de abstração alto – mais fácil gerar código.
- Conceito de objetos (métodos, construtores, herança, etc) já está presente no nível dos bytecodes.
- **Não é necessário se preocupar muito com otimização na geração dos bytecodes.**. Otimizações agressivas são feitas pelo JIT.
- Multiplaforma, imensa coleção de bibliotecas, etc.

Porque a escolha?

- Os próprios bytecodes possuem nível de abstração alto – mais fácil gerar código.
- Conceito de objetos (métodos, construtores, herança, etc) já está presente no nível dos bytecodes.
- **Não é necessário se preocupar muito com otimização na geração dos bytecodes.**. Otimizações agressivas são feitas pelo JIT.
- Multiplaforma, imensa coleção de bibliotecas, etc.

Dificuldades

- JVM foi inicialmente projetada **para rodar Java** (tipos estáticos estão presentes no nível de bytecodes, . . .) – existe uma dificuldade em compilar linguagens com características dinâmicas para a JVM.
- JDK7 incluiu uma nova instrução invokedynamic e um conjunto de mecanismos que facilitam a compilação de linguagens dinâmicas para a JVM, e, principalmente, permite o aumento do desempenho dessas linguagens (evitando o uso de *reflection*). Mesmo assim:
  - Os mecanismos para invocação dinâmica de métodos são extremamente genéricos, pois diferentes linguagens possuem diferentes mecanismos de *dispatch*.
  - JDK7 só foi lançada oficialmente há poucos meses. Durante o desenvolvimento da linguagem XBA, foi necessário trabalhar com várias versões beta e RCs, que tinham especificações 'instáveis'.

Dificuldades

- JVM foi inicialmente projetada **para rodar Java** (tipos estáticos estão presentes no nível de bytecodes, . . .) – existe uma dificuldade em compilar linguagens com características dinâmicas para a JVM.
- JDK7 incluiu uma nova instrução invokedynamic e um conjunto de mecanismos que facilitam a compilação de linguagens dinâmicas para a JVM, e, principalmente, permite o aumento do desempenho dessas linguagens (evitando o uso de *reflection*). Mesmo assim:
  - Os mecanismos para invocação dinâmica de métodos são extremamente genéricos, pois diferentes linguagens possuem diferentes mecanismos de *dispatch*.
  - JDK7 só foi lançada oficialmente há poucos meses. Durante o desenvolvimento da linguagem XBA, foi necessário trabalhar com várias versões beta e RCs, que tinham especificações 'instáveis'.

Dificuldades

- JVM foi inicialmente projetada **para rodar Java** (tipos estáticos estão presentes no nível de bytecodes, . . .) – existe uma dificuldade em compilar linguagens com características dinâmicas para a JVM.
- JDK7 incluiu uma nova instrução invokedynamic e um conjunto de mecanismos que facilitam a compilação de linguagens dinâmicas para a JVM, e, principalmente, permite o aumento do desempenho dessas linguagens (evitando o uso de *reflection*). Mesmo assim:
  - Os mecanismos para invocação dinâmica de métodos são extremamente genéricos, pois diferentes linguagens possuem diferentes mecanismos de *dispatch*.
  - JDK7 só foi lançada oficialmente há poucos meses. Durante o desenvolvimento da linguagem XBA, foi necessário trabalhar com várias versões beta e RCs, que tinham especificações 'instáveis'.

JavaCC

- Gera um analisador sintático recursivo descendente a partir de uma gramática
- Foi usado para gerar analisadores léxicos e sintáticos para a linguagem XBA.
- Também foi usada *jjtree*, que auxilia a geração da árvore sintática de um programa.

JavaCC

- Gera um analisador sintático recursivo descendente a partir de uma gramática
- Foi usado para gerar analisadores léxicos e sintáticos para a linguagem XBA.
- Também foi usada *jjtree*, que auxilia a geração da árvore sintática de um programa.

JavaCC

- Gera um analisador sintático recursivo descendente a partir de uma gramática
- Foi usado para gerar analisadores léxicos e sintáticos para a linguagem XBA.
- Também foi usada *jjtree*, que auxilia a geração da árvore sintática de um programa.

JavaCC


```
void FieldDefinition() #FieldDefinition:
{ Token member; }
{
    <MEMBER> (member = <ID>) { jjtThis.name(member); }
            <OP_EQ> Expression() <END_STATEMENT>
}


void MethodDefinition() #MethodDefinition:
{
    Token selector;
    Token id;
}
{
    (
        // Método com pelo menos um seletor com parâmetro e, opcionalmen
        (<METHOD>((selector = <BINARY_SELECTOR>)(id = <ID>) {
            jjtThis.addSelector(selector, id);
        })+ (selector = <UNARY_SELECTOR> {
            jjtThis.addUnarySelector(selector);
        })?)
        |
        // Método com apenas um único seletor unário
        (<METHOD>(selector = <UNARY_SELECTOR> {
            jjtThis.addUnarySelector(selector);
```

ASM

- Biblioteca para manipulação de bytecodes. Permite ler/modificar bytecodes de classes já existentes.
- Foi usada, principalmente, para *geraração* de bytecodes de classes do zero, por fornecer facilidades similiares às que um assembler fornece na geração de código de máquina:
  - Não é preciso se preocupar com o formato do arquivo .class.
  - Constantes são colocadas no *pool de constantes* da classe automaticamente.
  - *labels* para *goto*s, etc.
- Também foi usada para escrever os tipos inferidos de cada classe como anotações dentro dos arquivos .class

ASM

- Biblioteca para manipulação de bytecodes. Permite ler/modificar bytecodes de classes já existentes.
- Foi usada, principalmente, para *geraração* de bytecodes de classes do zero, por fornecer facilidades similiares às que um assembler fornece na geração de código de máquina:
  - Não é preciso se preocupar com o formato do arquivo .class.
  - Constantes são colocadas no *pool de constantes* da classe automaticamente.
  - *labels* para *goto*s, etc.
- Também foi usada para escrever os tipos inferidos de cada classe como anotações dentro dos arquivos .class

ASM

- Biblioteca para manipulação de bytecodes. Permite ler/modificar bytecodes de classes já existentes.
- Foi usada, principalmente, para *geraração* de bytecodes de classes do zero, por fornecer facilidades similiares às que um assembler fornece na geração de código de máquina:
  - Não é preciso se preocupar com o formato do arquivo .class.
  - Constantes são colocadas no *pool de constantes* da classe automaticamente.
  - *labels* para *goto*s, etc.
- Também foi usada para escrever os tipos inferidos de cada classe como anotações dentro dos arquivos .class

- Não há alguma forma direta de construir e chamar classes Java.
- Mas é possível gerar classes **XBA** válidas, fazendo classes em Java anotadas
- Uma aplicação desenvolvida (usando **ASM**) le as classes anotadas e gera classes **XBA** válidas.

```
@XBAClass(name = "Map")
public class XBAHash {
    private HashMap<XBAObject, XBAObject> map;

    @XBAMethod(selectors = { "init" }, type = MethodType.Initializer)
    public void initialize() {
        map = new HashMap<XBAObject, XBAObject>();
    }

    @XBAMethod(selectors = { "put", "with-value" })
    public XBAObject put(XBAObject key, XBAObject value) {
        map.put(key, value);
        return XBAUnit.instance;
    }

    @XBAMethod(selectors = { "get" })
    public XBAObject get(XBAObject key) {
        return map.get(key);
    }
}
```

- Não há alguma forma direta de construir e chamar classes Java.
- Mas é possível gerar classes **XBA** válidas, fazendo classes em Java anotadas
- Uma aplicação desenvolvida (usando **ASM**) le as classes anotadas e gera classes **XBA** válidas.

```
@XBAClass(name = "Map")
public class XBAHash {
    private HashMap<XBAObject, XBAObject> map;

    @XBAMethod(selectors = { "init" }, type = MethodType.Initializer)
    public void initialize() {
        map = new HashMap<XBAObject, XBAObject>();
    }

    @XBAMethod(selectors = { "put", "with-value" })
    public XBAObject put(XBAObject key, XBAObject value) {
        map.put(key, value);
        return XBAUnit.instance;
    }

    @XBAMethod(selectors = { "get" })
    public XBAObject get(XBAObject key) {
        return map.get(key);
    }
}
```

- Não há alguma forma direta de construir e chamar classes Java.
- Mas é possível gerar classes **XBA** válidas, fazendo classes em Java anotadas
- Uma aplicação desenvolvida (usando **ASM**) le as classes anotadas e gera classes **XBA** válidas.

```
@XBAClass(name = "Map")
public class XBAHash {
    private HashMap<XBAObject, XBAObject> map;

    @XBAMethod(selectors = { "init" }, type = MethodType.Initializer)
    public void initialize() {
        map = new HashMap<XBAObject, XBAObject>();
    }

    @XBAMethod(selectors = { "put", "with-value" })
    public XBAObject put(XBAObject key, XBAObject value) {
        map.put(key, value);
        return XBAUnit.instance;
    }

    @XBAMethod(selectors = { "get" })
    public XBAObject get(XBAObject key) {
        return map.get(key);
    }
}
```

Conteúdo

Considerações finais

- A parte que diz respeito ao sistema de tipos em si gerou um trabalho acima do esperado.
- A implementação do compilador em si está bem rudimentar e serve principalmente para demonstrar o funcionamento de um sistema de tipos estrutural.
- Por exemplo, mensagens de erro (sintáticas, do sistema de tipos, etc) não indicam com precisão o que está incorreto.

Apresentação está disponível em:
http://www.linux.ime.usp.br/∼hstagni/mac499

Considerações finais

- A parte que diz respeito ao sistema de tipos em si gerou um trabalho acima do esperado.
- A implementação do compilador em si está bem rudimentar e serve principalmente para demonstrar o funcionamento de um sistema de tipos estrutural.
- Por exemplo, mensagens de erro (sintáticas, do sistema de tipos, etc) não indicam com precisão o que está incorreto.

Apresentação está disponível em:
http://www.linux.ime.usp.br/∼hstagni/mac499

Considerações finais

- A parte que diz respeito ao sistema de tipos em si gerou um trabalho acima do esperado.
- A implementação do compilador em si está bem rudimentar e serve principalmente para demonstrar o funcionamento de um sistema de tipos estrutural.
- Por exemplo, mensagens de erro (sintáticas, do sistema de tipos, etc) não indicam com precisão o que está incorreto.

Apresentação está disponível em:
http://www.linux.ime.usp.br/∼hstagni/mac499

Considerações finais

- A parte que diz respeito ao sistema de tipos em si gerou um trabalho acima do esperado.
- A implementação do compilador em si está bem rudimentar e serve principalmente para demonstrar o funcionamento de um sistema de tipos estrutural.
- Por exemplo, mensagens de erro (sintáticas, do sistema de tipos, etc) não indicam com precisão o que está incorreto.

Apresentação está disponível em:
http://www.linux.ime.usp.br/∼hstagni/mac499