Análise Semântica

Guido Araújo
guido@ic.unicamp.br

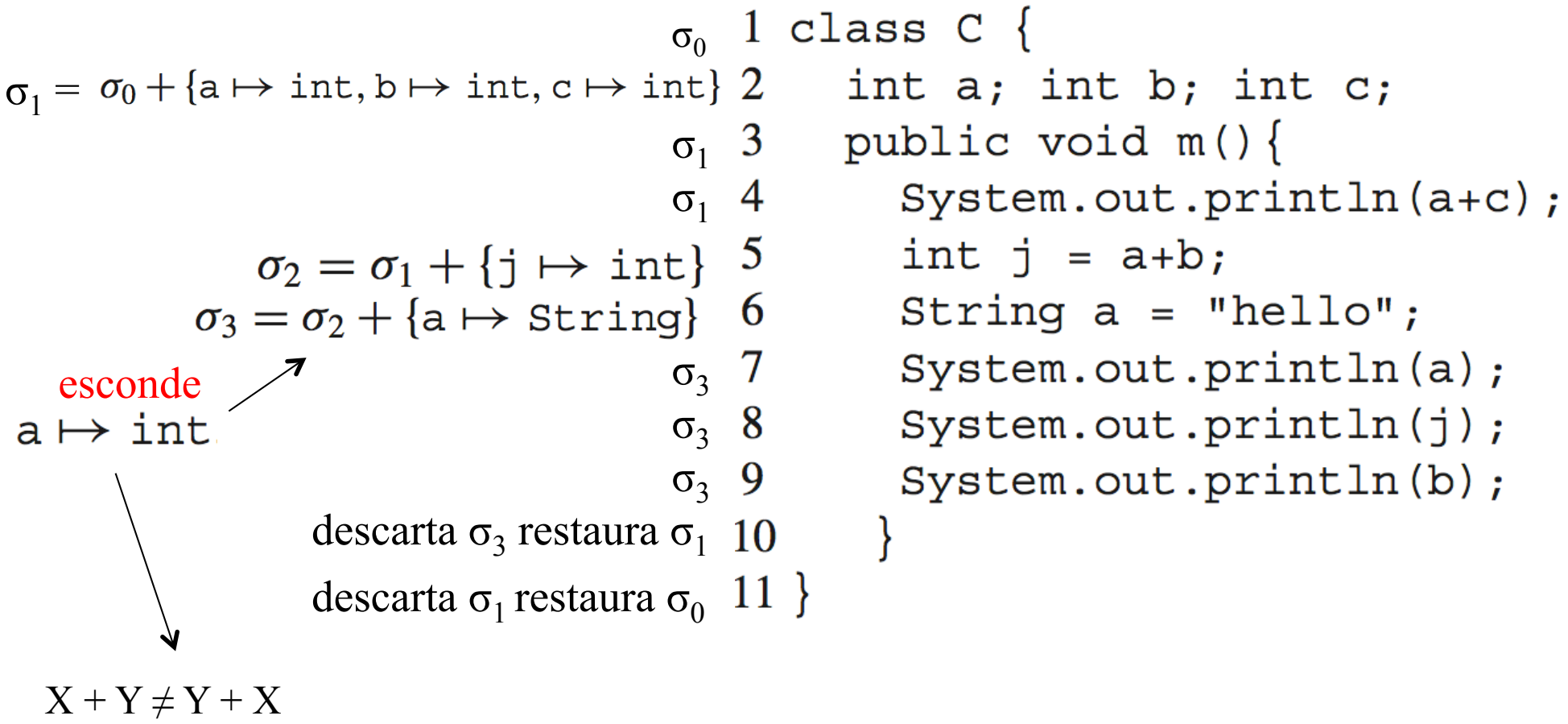
Tabela de Símbolos

- Associa variáveis a tipos (significados)

$$\sigma_1 = \{g \mapsto \texttt{string}, a \mapsto \texttt{int}\}$$

- Define o escopo em que a variável é visível
  - Parâmetros e variáveis locais de um método *m* em Minijava somente são visíveis dentro de *m*.
  - Campos e métodos de uma classe somente são visíveis dentro da classe
  - Para recuperar o escopo é preciso ter uma pilha de *undo*.

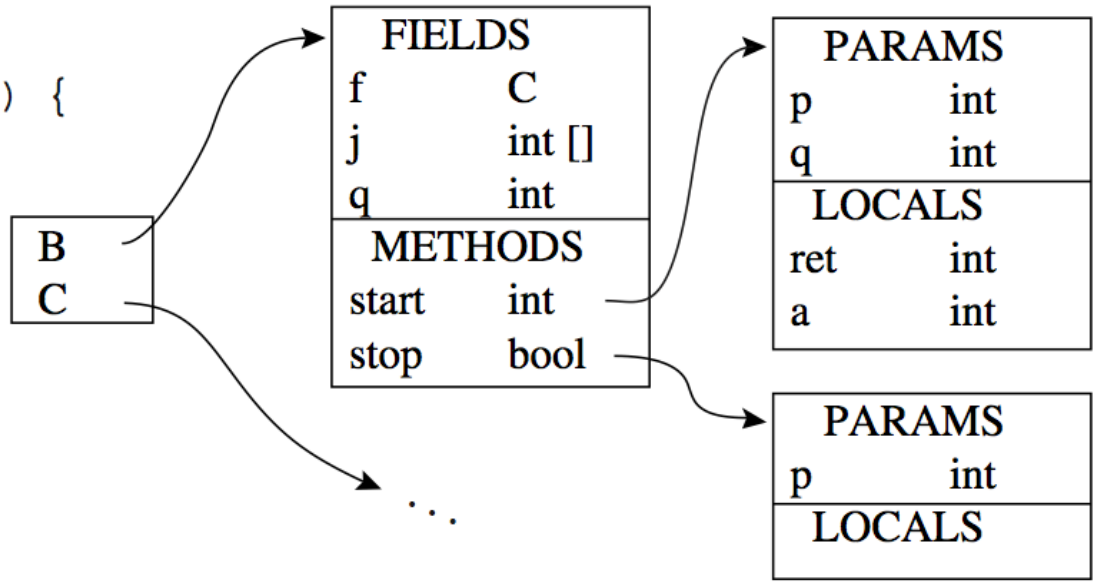
Construindo Tabela Símbolos



Exemplo

```
class B {
    C f;  int [] j;   int q;
    public int start(int p, int q) {
         int ret;   int a;
         /* ... */
         return ret;
    }
    public boolean stop(int p) {
              /* ... */
        retun false;
    }
}

class C {
   /* ... */
}
```



Como Implementar Escopo?

- Usando estilo funcional
  - Mantém cópias de σ_1, σ_2, etc…
- Usando estilo imperativo
  - Tem-se um único σ que vai se alterando em σ_1, σ_2, etc…
  - Mantém-se uma pilha de *undo*, e recupera-se o escopo quando desejado

Tabela de Símbolos

```
class Bucket {String key; Object binding; Bucket next;
        Bucket(String k, Object b, Bucket n) {key=k; binding=b; next=n;}
}

class HashT {
   final int SIZE = 256;
   Bucket table[] = new Bucket[SIZE];

   private int hash(String s) {
        int h=0;
        for(int i=0; i<s.length(); i++)
           h=h*65599+s.charAt(i);
        return h;
   }

   void insert(String s, Binding b) {
        int index=hash(s)%SIZE
        table[index]=new Bucket(s,b,table[index]);
   }
```

antes

$\sigma = \{a \mapsto T_1\}$

esconde a anterior

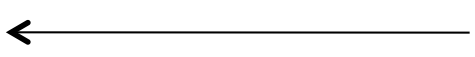
$\sigma + \{a \mapsto T_2\}$

Tabela de Símbolos

```
Object lookup(String s) {
      int index=hash(s)%SIZE
      for (Binding b = table[index]; b!=null; b=b.next)
        if (s.equals(b.key)) return b.binding;
      return null;
}

void pop(String s) {
      int index=hash(s)%SIZE
      table[index]=table[index].next;
}
}
```

depois



$\sigma = \{a \mapsto T_1\}$

Símbolos

- Problemas hash tables
  - Examina cada caracter da string para calcular o hash
  - Examina cada caracter para testar a string no bucket
  - Para evitar comparações desnecessárias converte todas as cadeias para um único objeto símbolo.

- Símbolos
  - Comparação para igualdade é rápida
  - Extrair um hash inteiro é rápido
  - Comparar dois símbolos para “greater-than” é rápido.

Tabela de Símbolos (pacote)

```
package Symbol;

public class Symbol {
  public String toString();
  public static Symbol symbol(String s);
}

public class Table {
  public Table();
  public void put(Symbol key, Object value);
  public Object get(Symbol key);
  public void beginScope();
  public void endScope();
  public java.util.Enumeration keys();
}
```

define início e final do escopo
gerenciando o *undo*
mantém *top* e *prevtop*

Tabela de Símbolos (implementação)

```
package Symbol;
public class Symbol {
  private String name;
  private Symbol(String n) {name=n; }
  private static java.util.Dictionary dict = new java.util.Hashtable();

  public String toString() {return name;}

  public static Symbol symbol(String n) {
        String u = n.intern();
        Symbol s = (Symbol)dict.get(u);
        if (s==null) {s = new Symbol(u); dict.put(u,s); }
        return s;
  }
}
```

gera símbolo único

Verificação em Duas Fases

- Fase 1
  - Construção tabela de símbolos (tipos básicos *int* e *boolean*)
  - Outros tipos são adicionados (novas classes, etc.)
- Fase 2
  - Verificação de tipos de *statements* e *expressions*
  - Tabela de tipos verificada para cada identificador achado
  - Adiciona finalmente na tabela de símbolos
- Preciso fazer em duas fases
  - Chamada de um método que ainda não foi definido.
  - Classes mutuamente dependentes

Verificação de Tipos em MiniJava

- Tabela de símbolos contém amarrações
  - Nome de variável ⟶ tipo
  - Nome de parâmetro formal ⟶ tipo
  - Nome de método ⟶ parâmetros, tipo de retorno e variáveis locais
  - Nome de classe ⟶ declaração de variáveis e métodos

Fase 1 Visitor: declarações

```
class ErrorMsg {
  boolean anyErrors;
  void complain(String msg) {
     anyErrors = true;
     System.out.println(msg);
  }
}

// Type t;
// Identifier i;
public void visit(VarDecl n) {

  Type t =  n.t.accept(this);
  String id =  n.i.toString();

  if (currMethod == null) {
      if (!currClass.addVar(id,t))
         error.complain(id + "is already defined in " + currClass.getId());
  } else if (!currMethod.addVar(id,t))
         error.complain(id + "is already defined in "
              + currClass.getId() + "." + currMethod.getId());
}
```

campo já visto

variável já vista

Fase 2 Visitor: stms e exps

- Exemplo: $e_1 + e_2$


```
// Exp e1,e2;
public Type visit(Plus n) {
  if (! (n.e1.accept(this) instanceof IntegerType) )
     error.complain("Left side of LessThan must be of type integer");
  if (! (n.e2.accept(this) instanceof IntegerType) )
     error.complain("Right side of LessThan must be of type integer");
  return new IntegerType();
}
```

Alguns outros casos….

- Assign
  - left-side same right-side
  - right-side subtype left-side
- Chamada de método
  - Buscar na tabela lista de parâmetros e tipo de retorno
  - Se *c.m()* e *c* é do tipo *C* então checar se existe método *m()* em *C*
  - Tipos dos parâmetros precisam casar com os tipos dos argumentos
  - O tipo da chamada recebe o tipo de retorno do método
- Outros
  - Checar no Java Language Specification

Tratando Erros

- Tentar recuperação

```
{int i = new C();
 int j = i + i;
 ...
}
```

i é tipo *C* !!

Ao invés de abortar, entrar *i* na tabela
como *int* para permitir verificar o resto do programa
Não executar as outras etapas do compilador !!