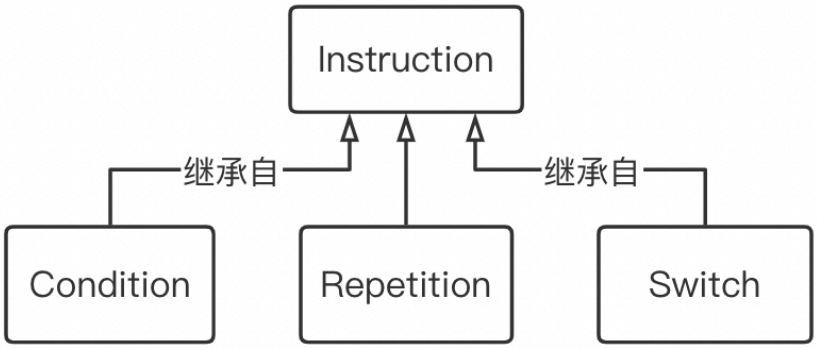
TIPS

类图的6种关系

【泛化关系 (Generalization)】

泛化关系是一种【继承】关系，表示一般与特殊的关系，它指定了子类如何特化父类的所有特征和行为。

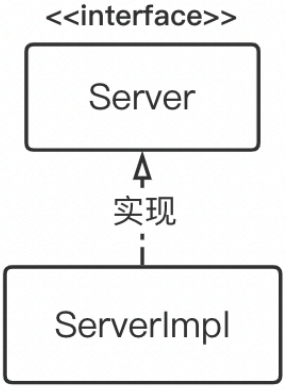
- 带空心三角箭头的实线，箭头指向父类



【实现关系 (Realization)】

实现关系是一种【类与接口】的关系，表示类是接口所有特征和行为的实现。

- 带空心三角箭头的虚线，箭头指向接口

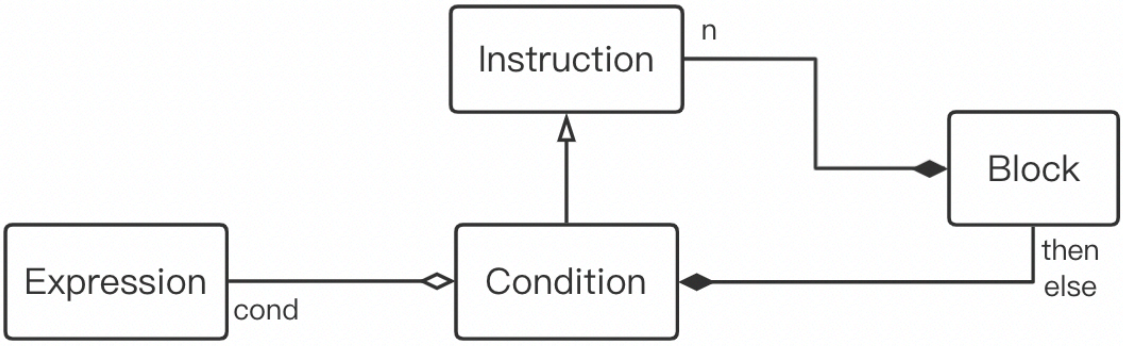


【聚合关系 (Aggregation)】

聚合关系是【整体与部分】的关系，且部分可以离开整体而单独存在。如车和轮胎是整体和部分的关系，轮胎离开车仍然可以存在。

聚合关系是关联关系的一种，是强的关联关系；关联和聚合在语法上无法区分，必须考察具体的逻辑关系。

- 代码体现：成员变量
- 带【空心菱形】的实心线，菱形指向整体



【组合关系 (Composition)】

组合关系是【整体与部分】的关系，但部分不能离开整体而单独存在。如公司和部门是整体和部分的关系，没有公司就不存在部门。

组合关系是关联关系的一种，是比聚合关系还要强的关系，它要求普通的聚合关系中代表整体的对象负责代表部分的对象的生命周期。

- 代码体现：成员变量
- 带【实心菱形】的实线，菱形指向整体（如上图）

【关联关系 (Association)】

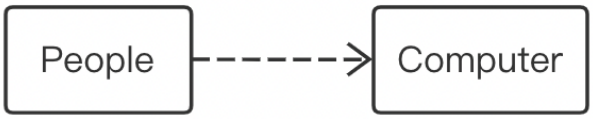
关联关系是一种【拥有】的关系，它使一个类知道另一个类的属性和方法。如：老师与学生，关联可以是双向的，也可以是单向的。双向的关联可以有两个箭头或者没有箭头，单向的关联有一个箭头。

- 代码体现：成员变量
- 带普通箭头的实心线，指向被拥有者

【依赖关系 (Dependency)】

依赖关系是一种使用的关系，即一个类的实现需要另一个类的协助，所以要尽量不使用双向的互相依赖.

- 代码表现：局部变量、方法的参数或者对静态方法的调用
- 带箭头的虚线，指向被使用者



程序实例

OCaML

1. type（以 if 为例）

```
(* ..........................................................  *)
(* unify : typeType * typeType -> typeType * bool             *)
(* unify 可以比较输入的2种类型，如果匹配则为 true，返回值是第一种类型  *)
(* 否则为 false，返回 ErrorType *)
(* ..........................................................  *)
```

```
type ast =
  | IfNode of ast * ast * ast;;

let rec type_of_expr env expr = match expr with
  | (IfNode econd ethen eelse) -> ruleIf env econd ethen eelse;;

ruleIf env econd ethen eelse =
  let tcond = (type_of_expr cond env) in
    let tthen = (type_of_expr ethen env) in
      let telse = (type_of_expr eelse env) in
        let _,tcond_is_bool = unify tcond BooleanType in
          let type_then_else,then_else_is_same_type = unify tthen telse in
            (if (tcond_is_bool && then_else_is_same_type)
              then type_then_else
              else ErrorType);;
```

2.value （以 if 为例）

```
let rec value_of_expr (expr,mem) env = match expr with
  | (IfNode econd ethen eelse) -> ruleIf env econd ethen eelse mem;;

ruleIf env econd bthen belse mem =
  let (cond_val,cond_mem) = (value_of_expr (econd,mem) env) in
    (match cond_val with
     | (BooleanValue rcond) ->
       (if (rcond) then
          (value_of_expr (bthen,cond_mem) env)
       else
          (value_of_expr (belse,cond_mem) env))
     | (ErrorValue _) as result -> (result,cond_mem)
     | _ -> ((ErrorValue TypeMismatchError),cond_mem))
```

TAM

详见 TD7: La machine TAM

```
int i = 6;
```

```
PUSH 1              /* 1. 规划用以操作变量的内存空间 */
LOADL 6             /* 2. 给变量赋值 */
STORE (1) 0[SB]     /* 3. 将(1)个变量存进地址0[SB]中 */
```

```
int y = i + 1;
```

```
PUSH 2              /* 1. 规划用以操作变量的内存空间 */
LOADL 1             /* 2. 加载其中一个操作数 */
LOAD (1) 0[SB]      /* 3. 从地址0[SB]中加载另一个操作数i */
SUBR IAdd           /* 4. 调用原语 IAdd */
STORE (1) 1[SB]     /* 5. 将得到的(1)个结果存到地址1[SB]中 */
```

TDS (参考 Projet)

collectAndBackwardResolve(TDS tds)

> 继承语义的属性以收集所有标识符声明并检查声明是否允许。
>
> Inherited Semantics attribute to collect all the identifiers declaration and check the declaration are allowed.

1. 在 Declaration 中，我们需要检查每一个 Expression 中的成员变量，如name，type，parameter 等是否在 TDS 中存在。以下是 TDS 中的方法：
   - boolean tds.contains(String name)：检查在该 Declaration 对象中的 name 属性是否在 tds 中存在
   - boolean tds.accepts(TDS tds)：检查在该 Declaration 对象是否可以注册到 tds 中
   - void tds.register(TDS tds)：在 tds 中注册该 Declaration 对象

   例 VariableDeclaration varDec implements Declaration, Instruction

```
public VariableDeclaration(String name, Type type, Expression expr);
public boolean collectAndBackwardResolve(TDS tds) {
    boolean tds_nameIsContrain = tds.contains(this.name);
    boolean tds_expr = this.expr.collectAndBackwardResolve(tds);
    Declaration varDec = new VariableDeclaration(this.name,this.type,this.expr);
    boolean tds_isAccepted = tds.accepts(varDec);
    if (tds_nameIsContrain) {
      Logger.error("Name is existed. ");
      return false;
    } else if (tds_isAccepted) {
      tds.register(varDec);
      tds_nameIsContrain = true;
    } else {
      return false;
    }
    return tds_nameIsContrain && tds_expr && tds_isAccepted;
  }
```

2. 在 Instruction 中，我们只需要自顶向下、递归地调用 Instruction 中成员的 collectAndBackwardResolve() 方法

   例 Condition c implements Instrution

```
public Condition (Expression condition, Block thenBranch, Block elseBranch);
public boolean collectAndBackwardResolve(TDS tds) {
    boolean tds_ok = this.condition.collectAndBackwardResolve(tds);
    tds_ok = tds_ok && this.thenBranch.collectAndBackwardResolve(tds);
    if (elseBranch != null) {
      tds_ok = tds_ok && this.elseBranch.collectAndBackwardResolve(tds);
    }
    return tds_ok;
  }
```

fullResolve(TDS tds)

> 继承语义属性以检查所有标识符是否已定义并将所有标识符使用与其定义相关联。
>
> Inherited Semantics attribute to check that all identifiers have been defined and associate all identifiers uses with their definitions.

Type

1. 根据 LL(1) 型文法规划结构，画出类图

1. $S \to B$
2. $B \to \{LI\}$
3. $LI \to I\ LI$
4. $LI \to \Lambda$
5. $I \to const\ T\ id = V$
6. $I \to \quad \ldots$
7. $T \to int$
8. $T \to \quad \ldots$
9. $E \to \quad \ldots$

- $\#5 \quad \{I.ast \to new\ Instruction(const.txt,\ T.ast,\ LI.ast)\}$