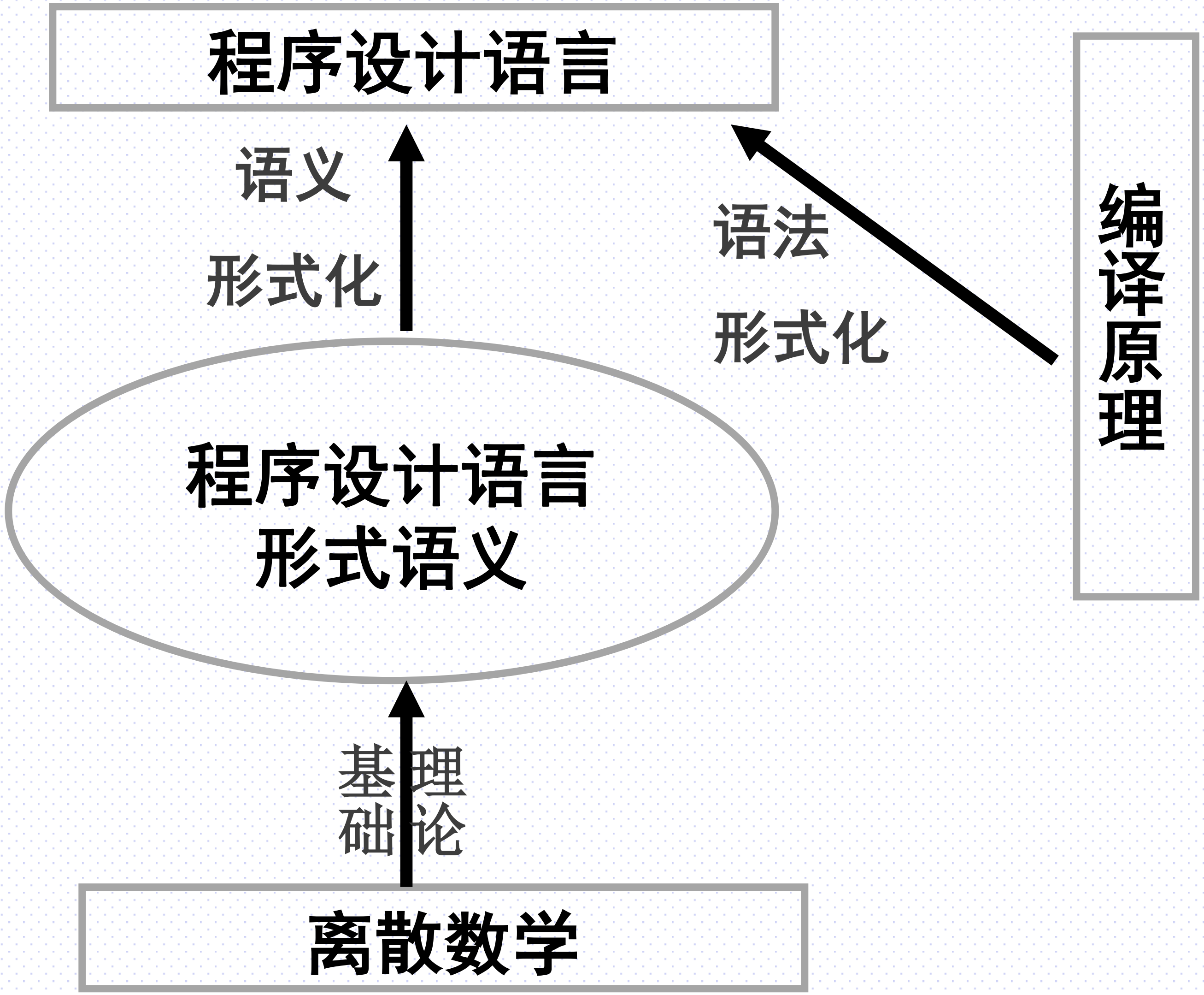
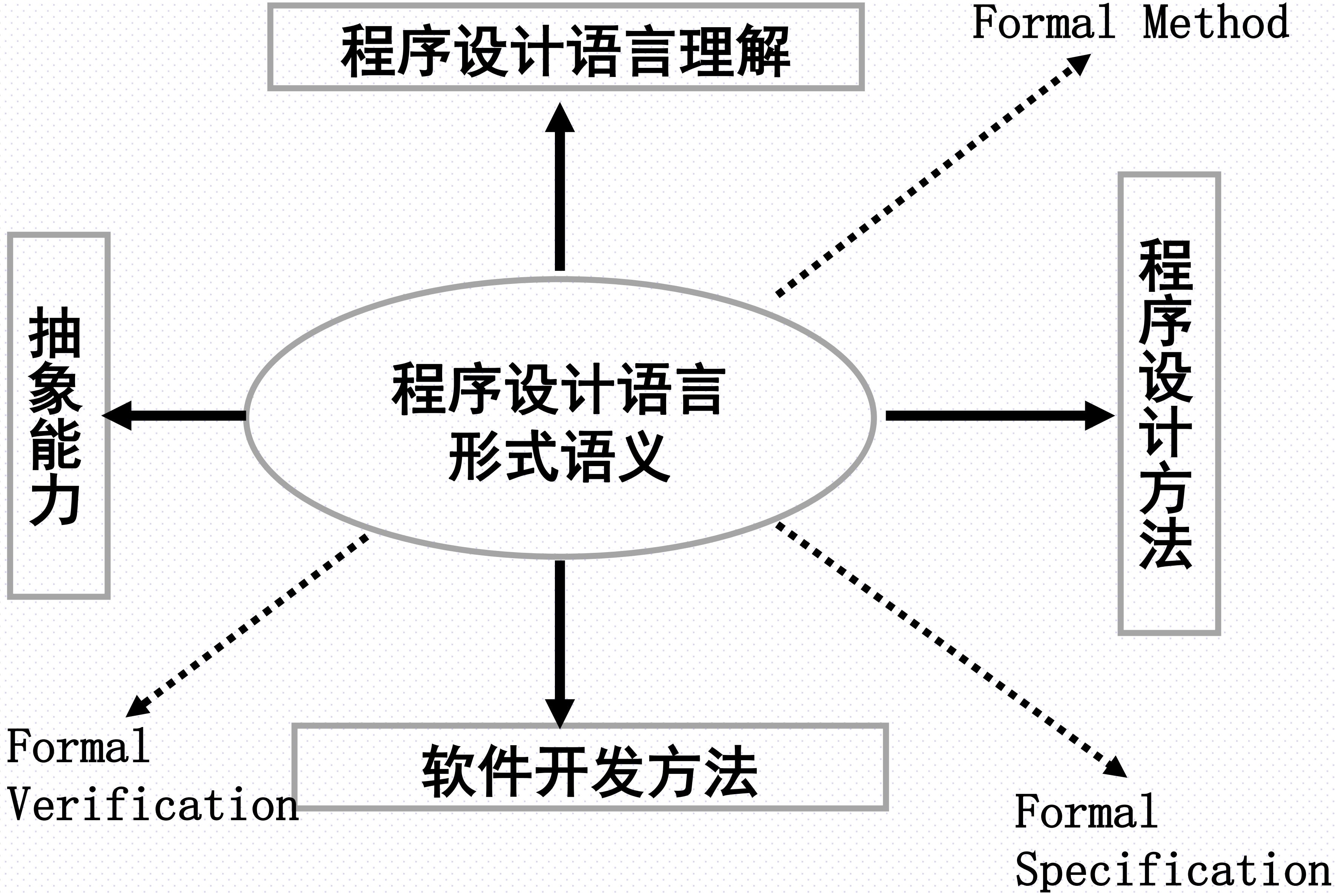
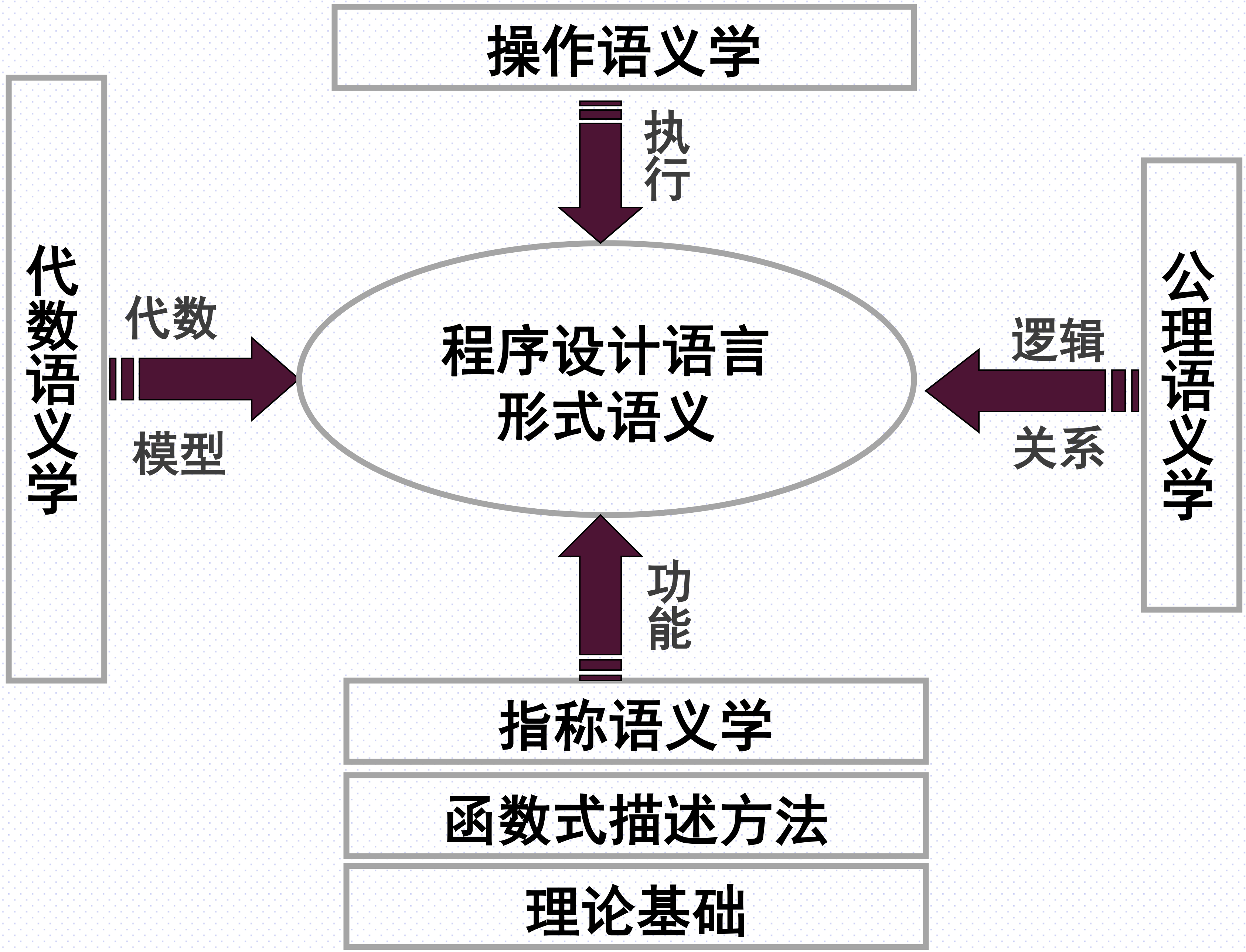

程序设计语言
语义
形式化
语法
形式化
编译原理
程序设计语言
形式语义
基础理论
离散数学



程序设计语言理解
Formal Method
抽象能力
程序设计语言形式语义
程序设计方法
Formal Verification
软件开发方法
Formal Specification



操作语义学
执行
代数语义学
代数
模型
程序设计语言
形式语义
逻辑
关系
公理语义学
功能
指称语义学
函数式描述方法
理论基础

第九章　指称语义的原理与应用

指称语义学是Christopher Strachey和Dana Scott在1970年提出的。

指称语义学的一个显著特征是：程序中的每一个短语(表达式、命令、声明等)都有其意义。它是与语言的语法结构平行的。每个短语的语义函数就是短语的指称意义。其现代名称为指称语义学。

16.1 指称语义原理

- 从数学的观点，一个程序可以看作是从输入到输出的映射P(I)→O，即输入域(domain)上的值，经过程序P变为输出域(range)的值。
  $\phi [\![p]\!] \rightarrow d \quad (p \in P, d \in D)$。
- 语义域D中的数学实体d，或以辅助函数表达的复杂数学实体d'，称为该短语的数学指称物，即短语在语义函数下的指称语义。
- 指称语义描述的是语义函数映射的后果，不反映如何映射的过程，更没有过程的时间性。而程序设计语言的时间性只能反映到值所表达的状态上。

• 语义函数和辅助函数

描述二进制数的语义

二进制数

```
    Numeral ::= 0                         (16.1-a)
              | 1                         (16.1-b)
              | Numeral 0                 (16.1-c)
              | Numeral 1                 (16.1-d)
```

我们给出求值的语义函数:将Numeral集合中的对象映射为自然数:

valuation: Numeral→Natural　　(16.2)

按语法的产生式，我们给出以下语义函数:

valuation 〖0〗= 0

valuation 〖1〗 = 1

valuation 〖N0〗 = 2 × valuation 〖N〗

//N∈Numeral

valuation 〖N1〗 = 2 × valuation 〖N〗+ 1

valuation〖1101〗 = 2 × valuatioin 〖110〗+ 1
= 2 × (2 × valuation 〖11〗) + 1
= 2 × (2 ×(2 × valuation 〖1〗+ 1)) + 1
= 2 × (2 ×(2 ×1 + 1)) + 1
= 13

计算器命令的语义描述

计算器命令的抽象语法：

Com ::= Expr= (16.3)
Expr ::= Num (16.4-a)
| Expr + Expr (16.4-b)
| Expr - Exp (16.4-c)
| Expr * Expr (16.4-d)
Num ::= Digit | Num Digit (16.5)
Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 (16.6)

execute : Com → Integer
evaluate: Expr → Integer
sum : Integer × Integer → Integer
difference : Integer × Integer → Integer
product : Integer × Integer → Integer

以下定义每个短语的语义函数:

execute 〖C〗 = execute 〖E=〗 = evaluate 〖E〗

其中C∈Com，E∈Expr。

evaluate 〖N〗= valuation 〖N〗 (N∈Num)

evaluate 〖E1 + E2〗 = sum (evaluate 〖E1〗, evaluate 〖E2〗)

evaluate 〖E1 - E2〗 = difference (evaluate 〖E1〗, evaluate 〖E2〗)

evaluate 〖E1 * E2〗 = product (evaluate 〖E1〗, evaluate 〖E2〗)

再定义Num的两个表达式:

valuation 〖D〗 = D’ (D∈Digit, D’ ∈Natural)

valuation 〖ND〗= 10 × valuation 〖N〗+ D’

execute 〖40-3*9=〗

=evaluate 〖40-3*9〗

=product (evaluate〖40-3〗, evaluate〖9〗)

=product (difference (evaluate 〖40〗, evaluate 〖3〗), evaluate〖9〗)

=product (difference (valuation〖40〗, valuation〖3〗), valuation〖9〗)

=product (difference (40, 3), 9)

=333

16.1.2 语义域

- **基本域**

  Character / Integer / Natural / Truth-Value / Unit

  用户可定义枚举域，以及以基本域构造的复合域。

- **笛卡儿积域**

  D×D'　元素为对偶(x，x')其中x∈D，x'∈D'。

  D1×D2×…Dn元素为n元组(x1，x2，…，xn)，其中xi∈Di。

- **不相交的联合域**

  D+D'　元素为对偶(left x，right x')其中x∈D，x'∈D'。

  shape=rectangle(Real×Real) + circle Real + point

- **函数域**

  D→D’　例如Integer→Even。

  f(v) →⊥　　　偏函数，v∈V

  f(⊥)→⊥　　　严格的偏函数

  f(⊥)→v　　　非严格函数

  偏函数域上元素间具有偏序关系，偏序关系‘≤’的性质是:

- D域若具偏序性质，它必须包含唯一的底元素，记为⊥，且⊥≤d，d为D中任一元素。通俗解释是d得到的定义比⊥多。⊥是不对应任何值的‘值’。

- 若 x，y ∈D，x≤y此二元素具有偏序关系‘≤’，即y得到的定义比x多。这一般就复合元素而言，即x中包含的⊥比y多。

- 若x，y，z∈D，则偏序关系‘≤’必须是:

  [1] 自反的，即有x≤x;

  [2] 反对称的，即若x≤y，y≤x，必然有x=y;

  [3] 传递的，即若x≤y，y≤z，必然有x≤z。

- **序列域**

序列域D*中的元素是零个或多个选自域D中的元素有限序列，或为nil元素，或为x・s的序列

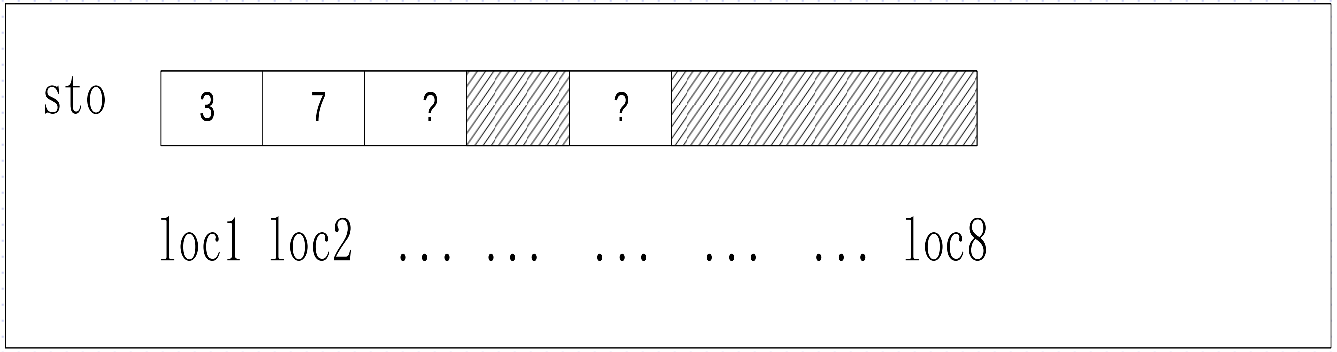
nil // 一般写法是“ ”

‘a’ ・nil // 一般写法是“a”

‘B’ ・ ‘u’ ・ ‘s’ ・ ‘y’ ・ nil //一般写法是“Busy”

16.1.3 命令式语言的特殊域

- **存储域**



Store = Location → (stored Storable + undefined + unused)　　　(16.15)

```
empty-store : Store                                          (16.16)
    allocate    : Store →Store × Location                    (16.17)
    deallocate  : Store ×Location → Store                    (16.18)
    update      : Store × Location × Storable→Store          (16.19)
    fetch       : Store × Location→Storable                  (16.20)
empty_store =  λloc.unused
allocate  sto =
    let loc = any_unused_location (sto) in (sto [loc→ undefined], loc)
deallocate (sto, loc) = sto [loc → unused]
    update (sto, loc, stble) = sto [loc→stored stble]
    fetch (sto, loc) =
      let stored_value (stored stble) = stble
          stored_value (undefined) = fail
          stored_value (unused) = fail
      in
          stored-value (sto(loc))
```

- **环境域**

```
Environ = ldentifier→(bound Bindable + unbound)
empty-environ : Environ
    bind       : ldentifier×Bindable → Environ
    overlay    : Environ×Environ → Environ
find           : Environ×ldentifier→Bindable
enpty-environ = λI. unbound
    bind (I, bdble) = λI'. if I'=I then bound bdble else unbound
    overlay (env', env) = λI. if env' (I)/=unbound then env' (I) else env (I)
    find (env, I) =
          let bound_value (bound  bdble) = bdble
              bound_value (unbound) = ⊥
          in
            bound_value (env (I))
```

16.2 指称语义示例

- **过程式小语言 IMP**

抽象语法是:

```
    Command   ::= Skip
                | ldentifier := Expression
                | let Declaration in Command
                | Command; Command
                | if Expression then Command else Command
                | while Expression do Command
  Expression  ::= Numeral
                | false
                | true
                | Ldentifier
                | Expression + Expression
                | Expression < Expression
                | not Expression
                | ...
 Declaration  ::= const ldentifier = Expression
                | var ldentifier : Type_denoter
 Type_denoter ::= bool
                | int
```

IMP的语义域、语义函数和辅助函数

Value = truth_value Truth_Value + integer Integer

Storable = Value

Bindable = value Value + variable Location

execute: Command →(Environ→Store→Store)

execute〖C〗 env sto = sto’

evaluate: Expression → (Environ→Store →Value)

evaluate 〖E〗 env sto= …

elaborate: Declaration→ (Environ→Store→Environ×store)

elaborate 〖D〗 env sto =

辅助函数有如前所述的empty-environ，find，overlay，bind，empty-store，allocate，deallocate，update，fetch。以及sum，less，not等辅助函数。此外，再增加一个取值函数:

coerce: Store×Bindable→Value

coerce (sto，find (env，I))

= val

= fetch (sto，loc)

IMP的指称语义

```
execute 〖Skip〗 env sto = sto
execute 〖I:= E〗 env sto =
     let val = evaluate E env sto in
     let variable loc = find (env, I) in
     update(sto, loc, val)
execute 〖let D in C〗 env sto =
     let (env', sto') = elaborate D env sto in
     execute C (overlay (env', env)) sto'
```

```
execute 〖C1; C2〗 env sto =
      execute C2 env (execute C1 env sto)
execute 〖 if E then C1 else C2〗 env sto =
        if evaluate E env sto = truth_value true
        then execute C1 env sto
        else execute C2 env sto
execute 〖while E do C〗=
       let execute_while env sto =
         if evaluate E env sto = truth_value true
         then execute_while env (execute C env sto)
         else sto
       in
       execute_while
```

```
elaborate 〖 const I = E 〗 env sto =
          let val = evaluate E env sto in
          (bind (I, value val), sto)

elaborate 〖 var I:T〗 env sto =
          let (sto’, loc)= allocate sto in
          (bind (I, variable loc), sto’)
```

16.3 程序抽象的语义描述

- **函数抽象**

```
Function = Argument→Value
Function = Argument→Store→Value
bind_parameter: Formal_Parameter→(Argument→Environ)
give_argument : Actual_Parameter→(Environ→Argument)
```

- **扩充IMP语法**

```
Command ::= …
           | Identifier (Actual_Parametor)
Expression ::= …
           | Identifier (Actual_Parmenter)
Declaration ::= …
           | func Identifier (Formal_Parameter)  is Expression
           | proc ldentifier (Formal_paramenter)  is Command
Formal_Parameter ::= const Identifier: Type_Denoter
Actual_parameter ::= Expression
Argument = Value
Bindable = value Value + variable Location + function Function
```

- **写IMP函数的指称语义**

bind-parameter 〖I:T〗 arg = bind (I, arg)

give-argument 〖E〗 env = evaluate E env

函数调用的语义等式如下:

```
evaluate 〖I(AP)〗 env =
let function func = find (env, I) in
let arg = give_argument AP env  in
func arg

elaborate 〖fun I(FP) is E 〗 env =
   let func arg =
      let parenv = bind_parameter FP arg in
      evaluate E (overlay (parenv, env ))
   in
 (bind (I, function func))
```

- **过程抽象**

```
Procedure = Argument→Store→Store
Argument = Value
Bindable = value Value + variable Location+functionFunction +procedure Procedure
execute 〖 I(AP) 〗 env sto=

let procedure proc = find (env, I) in
     let arg = give_argument AP env sto in
     proc arg sto
elaborate 〖proc I(FP) is C 〗 env sto =
     let proc arg sto’ =
         let parent = bind-parameter FP arg in
         execute C (overlay (parenv env)) sto’
     in
    (bind (I, procedure proc), sto)
```

- 参数机制的语义描述

  --- 常量和变量参数

  先细化参数定义语法

```
Formal-Parameter ::= const Identifier: Type_denoter
                   | var Identifier : Type_denoter
Actual-P arameter ::= Expression
                   | var Identifier
bind_parameter : Formal_parameter→(Argument→Environ)
give_parameter : Actural_Parameter→(Environ→Store→Argument)
```

  形参的语义等式是:

```
bind_parameter 〖const I:T〗 (value val) = bind (I, value val)
bind_parameter 〖var I:T〗 (variable loc)= bind(I, variable loc)
```

  实参的语义等式是:

```
give_argument 〖E〗 env sto = value (evaluate E env sto)
give_argument 〖var I〗 env sto =
   let variable loc = find (env, I) in
     variable loc
```

--- 复制参数机制

```
Formal_Parmeter ::= value Identifier: Type_denoter
                  | result Identifier : Type_denoter
Actual_Parameter ::= Expression
                  | var Identifier
copy_in: Formal_Parameter→(Argument→Store→Environ×Store)
copy_in 〖value I:T〗 (value val) sto =
       let (sto’, local) = allocate sto in
       (bind (I, variable local), update (sto’, local, val))
copy-in 〖 result I:T〗 (variable loc) sto=
       let (sto’, local)= allocate sto in
       (bind (I, variable local), sto’)

copy_out: Formal_Parameter→(Environ→ Argument→Store→Store)
   copy_out 〖 value I:T〗 env (vlaue val) sto = sto
   copy_out 〖result I:T〗 env (variable loc) sto =
      let variable local = find (env, I) in
      update (sto, loc, fetch (sto, local))
```

过程声明的语义等式作以下修改:

```
elaborate〖proc (FP) is C 〗 env sto=
   let proc arg sto'=
       let (parenv, sto″) copy_in FP arg sto' in
       let sto''' = execute C (overlay (parenv, env )) sto″ in
           copy_out FP parenv arg sto'''
   in
      (bind (I, procedure proc), sto)
```

--- 多参数

```
Function = Argument*→Store→Value
Procedure = Argument* →Store→Store
bind_parameter : Formal_Parameter_List→(Argument*
                  →Environ)
give_argument:Acrual_Parameter_List→(Environ
                  →Store →Argament*)
```

--- 递归抽象

递归函数声明的语义等式如下:

```
    elaborate〖 fun I (FP) is E〗 env=
       let func arg =
         let env’=overlay (bind (I, function func), env) in
         let parenv = bind-parameter FP arg in
            evaluate E (overlay (parenv, env’))
       in
       bind (I, function func)
```

16.4　复合类型

➢最简单的复合变量的语义描述

暂不考虑函数和过程抽象，只增加最简单的复合量对偶(A:T1,B:T2).先扩充抽象语法:

```
Command ::= …
              | V_name := Expression
              |  …
Expression ::= …
              | V_name
              |  …
              |  (Expression,Expression)
V-name  ::= Identifier
                | fst V_name    //相当于V(1)
                | snd V_name    //相当于V(2)
Type_denoter ::= bool  | int
                |(Type_denoter, Type_denoter)
```

对偶值本身是一个域:
Pair_Value = Value×Value
对偶变量的域：
Pair_Variable = Variable×Variable
Value = truth_value Truth_Value + integer Integer + pair_value Pair_Value
Storable = truth_value Truth_Value + integer Integer
Variable = simple_variable Location + pair_variable Pair_Variable

辅助函数：

fetch_variable: Store×Variable→ Value
update_variable: Store×Variable×Value→ Store

```
fetch_variable(sto， simple_variable loc) = fetch(sto， loc)
fetch_variable(sto， pair_variable (var1， var2)) =
        pair_value(fetch_variable(sto， var1),fetch-variable (sto， var2))
update_variable(sto,simple_variable loc， stble ) =
             update (sto,loc,stble)
update_variable (sto,pair_variable (var1,var2), pair_value (val1,val2))=
            let sto'=update_variable (sto， var1,val1) in
                update_variable (sto',var2,val2)
```

增加识别(identify)和分配变量存储(allocate_variable)的语义函数：

```
identify: V-name → (Environ→ Value_or_Variable)
Value_or_Variable = value Value + variable Variable

    identify〖I〗 env = find(env, I)
    identify〖fst V 〗env =
       let first (value (pair_value (val1, val2))) = value val1 |
          first (variable (pair_variable (var1, var2))) = variable var1
      in
      first (identify V env)//辅助函数first将对偶值或对偶变量映射为它的第一子域。
```

赋值语句语义等式：

```
execute 〖V:= E〗 env sto =
      let val = evaluate E env sto in
      let variable var = identify V env in
      update_variable (sto, var, val)
evaluate 〖V〗 env sto=
       coerce (sto, identify V env)
coerce: Store ×Value_or_Variable→Value
    coerce (sto, value val ) = val
coerce (sto, variable var) = fetch_variable (sto, var)
```

```
allocate_variable: Type_denoter→Allocator
Allocator = Store→ Store×Variable
```

例：为类型指明符bool分配存储的语义是:

```
    allocate_variable 〖bool〗 sto=
      let (sto’, loc) = allocate sto in
         (sto’, simple_variable loc)
```

为对偶指明符分配存储的语义是:

```
    allocate_variable〖(T1, T2)〗 sto=
      let (sto’, var1) = allocate_variable T1 sto in
      let (sto’ , var2) = allocate_variable T2 sto’ in
      (sto’ , pair_variable (var1, var2))
```

变量声明的语义:

```
    elaborate〖var I:T〗 env sto=
       let (sto’, var) = allocate_variable T sto in
       (bind(I, var), sto’)
```

➢数组变量的语义描述(参考教材)

16.5 程序失败的语义描述

```
sum: Integer × Integer→ Integer
   sum (int1, int2) = if abs(int1+int2) <=maxint
                      then int1+int2
                      else ⊥
   sum (⊥, int2)= ⊥
   sum (int1, ⊥) = ⊥
evaluate  ⟦E1 + E2⟧  env sto =
      let integer int1 = evaluate E1 env sto in
      let integer int2 = evaluete E2 env sto in
      integer (sum (int1, int2))
```

16.6 指称语义应用

- **指称语义用于设计语言**

  为一个程序设计语言写指称语义的步骤是:

- 分析(所设计的)程序设计语言的规格说明写出抽象语法。

- 定义该语言的指称域，并为这些域定义恰当的辅助函数与模型值上的操作。

  建立语义函数。为抽象语法中的每个短语(即短语类)指定一个域(语义函数的　输入域)，定义输入域到其指称域的语义函数。

- 为每一短语类写出语义等式。

16.6.2 指称语义用于程序性质研究

● 上下文约束的静态描述

在程序设计语言的文法产生的所有句子之中只有一部分是良定义的。语法往往不能给出明确的表示，要依靠上下文约束。

用指称语义的方法描述程序设计语言的上下文约束要建立类型环境的概念。语言中各类型之总称即为Type域。例如，在前述IMP语言中类型域是：

Type=truth_type + integer_type + var_type + error_type

Type_Environ = Identifier→(bound Type + unbound)

equivalent: Type×Type→Truth_Value

可测试两种类型是否等价。

constrain: Command→(Type_Environ→Truth_Value)

检查命令在类型环境中是否遵从约束，即是否良定义的。

typify: Expression→(Type_Environ→Value_Type)

验明表达式的类型，即在类型环境中的具体类型。

declare : Declaration→(Type_Environ→Truth_Value×Type_Environ)

在类型环境中给出声明是良定义的真值，以及所产生的类型束定。

type_denoted_by: Type_Denoter→Value_Type

产生类型指明符的真实类型。类型环境域有以下辅助函数:

empty_environ : Type_Environ
bind : ldentifier × Type →Type_Environ
overlay: Type_Environ×Type_Environ→Type_Environ
find: Type_Environ×Identifier→Type

● 程序推理

C; ship ≡ C。要证明相等，即指出两端指称一样即可:

```
      execute 〖C; skip〗 env sto
      = execate 〖skip〗 env (execute C
env sto)
      = execute C env sto
```

将域的各等式也转成ML的datatype定义:

```
type Location = int;
datatype Value=
    truthvalue of bool
    |integer of int;
type Stroeable = Value;
datatype Bindable =
    value of Value
    | variable of Location;
```

再写出具体函数定义:

```
fun
  execute (skip) env sto = sto
  |execute (IbceomesE(I, E)) env sto =
   let val val' = evaluate E env sto in
     let val variable loc = find (env, I) in
         update (sto, loc, val')
     end
   end
  | execute (letDinC (D, C)) env sto =
   let val (env', sto') =
            elaborate D env sto in
      execute C (overlay (env', env)) sto'
   end
```

16.6.3 语义原型

先将抽象语法改写为ML的datatype 定义:

```
type Identifier = string
and Num eral = string;
datatype Command =
      skip
     | IbecomesE of Identifier * Expression
     | letDinC of Declaraton * Command
     | CsemicolonC of Command * Command
     | ifEthenCelseC of Expressiion * Command * Command
     | whileEdoC of Expression * Command
 and Expression =
       num of Numeral
     | flase'
     | true'
     | ide of Identifier
     | EplusE of Expression * Expression
 and Declaration=
        constIisE of Ldentifier * Expression
     | varIcolonT of Ldentifier* Typerdenoter
 and Typedenoter=
      bool'
     | int'
```

将域的各等式也转成ML的datatype定义:

```
type Location = int;
datatype Value=
    truthvalue of bool
     | integer of int;
type Stroeable = Value;
datatype Bindable =
    value of Value
     |  variable of Location;
```

再写出具体函数定义:

```
fun
  execute (skip) env sto = sto
 | execute (IbceomesE(I, E)) env sto =
   let val val' = evaluate E env sto in
     let val variable loc = find (env, I) in
         update (sto, loc, val')
     end
   end
 |  execute (letDinC (D, C)) env sto =
   let val (env', sto') =
             elaborate D env sto in
      execute C (overlay (env', env)) sto'
   end
```

```
| execute (CsemicolonC (C1, C2)) env sto =
    execute C2 env (execate C1 env sto)
| execute (if E then C else C (E, C1, C2)) env sto =
   if valuate E env sto = truthvalue true
   then execute C1 env sto
   else execute C2 env sto
| execute (whileEdoC (E, C))=
    let fun executewhile env sto =
            if evaluate E env sto = truthvalue true
            then executewhile env (execute C env sto )
            else sto
    in
     executewhile
    end
and
  evaluate (num N) env sto = integer (valuation N)
 |evaluate (false') env sto = truthvalue false
 |evaluate (true') env sto = truthvalue true
 |evaluate (ide I) env sto = coerce (sto, find (env, I ))
```

```
 |evaluate (EplusE( E1, E2 )) env sto =
      let val integer int1 = evaluate E1 env sto  in
         let val integer int2 = evaluate E2 env sto in
                 integer (sum ( int1, int2 ) )
         end
       end
 |...
and
elaborate (constIisE ( I, E )) env sto=
   let val val'=evaluate E env sto in
       (bind (I, value val' ), sto)
   end
 |elaborate (varIcolonT( I, T )) env sto=
     let val (sto', loc)=allocate sto in
           ( bind ( I, variable loc ), sto')
     end
and
  valuation (N) =
      integer (stringtoint N)
```

以上按IMP 抽象语法套写语义函数execute，evaluate，elaborate. 还要把辅助函数改写为ML:

```
fun
  coerce ( sto, value val')= val'
   | coerce ( sto, vatiable loc ) = fetch ( sto, loc )
```

设置初始条件运行ML程序
有了以上定义，即可运行抽象语法树的ML解释器，例如:

```
val env0=...;                                       //初始环境
val sto0=...;                                       //初始存储
val prog=...;                                       //一条IMP命令的抽象语法树
execute prog env0 sto0;
```

作业

– 在IMP语言中增加取值和寻址表达式、指针变量初始化命令，其语法进行了如下扩充：

Declaration ::= const Identifier = Expression

| *Identifier = Expression

Expression :: = *Identifier

| &Identifier

```
evaluation 〖*I〗 env sto =
        let loc = find( env, I) in
                let loc1 = fetch (sto, loc) in
                        sto(loc1)

evaluation 〖&I〗 env sto =
                        find(env, I)

execute 〖*I = E〗 env sto =
    let val = evaluate E env sto in
    let loc = find( env, I) in
                let loc1 = fetch (sto,loc) in
                        sto(loc1)
    update(sto, val, loc)
```

大作业的要求

- 组队：每组不多于3人
- 提交设计报告

1. 新定义语言的背景和目标

    设计驱动，基础范型，参考语言及其不同点，围绕设计准则方面的考虑

2. 语法设计

    语言要素，静态/动态、编译/解释、跨平台等方面的考虑；数据类型、关键字、Token对象等词法规则考虑；BNF（举例用图来表达）、抽象语法树、语法分析等的设计考虑

3. 涉及范型的设计

    控制流相关的设计（分支、迭代等）；对象、并发机制、闭包等

4. 典型语言机制的语义描述（举例说明）
5. 与对标语言在实现上的差异说明

    语言差异，运行差异

6. 验证与测试

提交时间与形式：12月16日23:00前提交，在课程中心上提交pdf报告，准备评优同学还需要准备ppt。

大作业评优的条件

- 小组不多于2人
- 最多给6~7组（选课人数的10%左右）
- 考试时完成B卷
- 考核标准：
  - 在普通作业标准基础上：
  - 课堂讲解（PPT）
  - 可运行、有测试样例
- 大作业派前8的组别都要准备ppt在研讨课上讲，12月19日出通知，确定哪些组在12月21日的研讨课上讲台（形式为每组讲解10~12分钟，回答问题5~8分钟）。
- 台下同学需随时准备提问，有质量的提问将被记录并获得大作业加分。