# 程序语言语法的 Coq 定义

## 1 一个极简的指令式程序语言：SimpleWhile

下面是 SimpleWhile 语言中程序表达式的语法。

```
EI :: = N | V | EI + EI | EI - EI | EI * EI
EB :: = TRUE | FALSE | EI < EI | EB && EB | ! EB
```

SimpleWhile 语言的程序语句包括空语句、变量赋值语句、顺序执行、if 语句与 while 语句。

```
 C :: = SKIP |
        V = EI |
        C; C |
        if (EB) then { C } else { C } |
        while (EB) do { C }
```

在 Coq 中，我们就用字符串表示变量名，

```
Definition var_name: Type := string.
```

并且使用 Coq 归纳类型定义表达式和语句的语法树。

```
Inductive expr_int : Type :=
  | EConst (n: Z): expr_int
  | EVar (x: var_name): expr_int
  | EAdd (e1 e2: expr_int): expr_int
  | ESub (e1 e2: expr_int): expr_int
  | EMul (e1 e2: expr_int): expr_int.
```

```
Inductive expr_bool: Type :=
  | ETrue: expr_bool
  | EFalse: expr_bool
  | ELt (e1 e2: expr_int): expr_bool
  | EAnd (e1 e2: expr_bool): expr_bool
  | ENot (e: expr_bool): expr_bool.
```

```
Inductive com : Type :=
  | CSkip: com
  | CAsgn (x: var_name) (e: expr_int): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr_bool) (c1 c2: com): com
  | CWhile (e: expr_bool) (c: com): com.
```

在 Coq 中，可以利用 `Notation` 使得这些表达式和程序语句更加易读，下面是一些使用 `Notation` 的例子（`Notation` 的具体定义详见 Coq 源代码）。

```
Check [[1 + "x"]].
Check [["x" * ("a" + "b" + 1)]].
Check [[1 + "x" < "x"]].
Check [["x" < 0 && 0 < "y"]].
Check [["x" = "x" + 1]].
Check [[while (0 < "x") do { "x" = "x" - 1}]].
```

## 2 更多的程序语言：While 语言

在许多以 C 语言为代表的常用程序语言中，往往不区分整数类型表达式与布尔类型表达式，同时表达式中也包含更多运算符。例如，我们可以如下规定一种程序语言的语法。

```
E :: = N | V | -E | E+E | E-E | E*E | E/E | E%E |
       E<E | E<=E | E==E | E!=E | E>=E | E>E |
       E&&E | E||E | !E
```

```
C :: = SKIP |
       V = E |
       C; C |
       if (E) then { C } else { C } |
       while (E) do { C }
```

下面依次在 Coq 中定义该语言中的二元运算符和一元运算符。

```
Inductive binop : Type :=
  | OOr | OAnd
  | OLt | OLe | OGt | OGe | OEq | ONe
  | OPlus | OMinus | OMul | ODiv | OMod.
```

```
Inductive unop : Type :=
  | ONot | ONeg.
```

然后再定义表达式的抽象语法树。

```
Inductive expr : Type :=
  | EConst (n: Z): expr
  | EVar (x: var_name): expr
  | EBinop (op: binop) (e1 e2: expr): expr
  | EUnop (op: unop) (e: expr): expr.
```

最后是程序语句的抽象语法树。

```
Inductive com : Type :=
  | CSkip: com
  | CAsgn (x: var_name) (e: expr): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr) (c1 c2: com): com
  | CWhile (e: expr) (c: com): com.
```

# 3 更多的程序语言：WhileDeref

下面在 While 程序语言中增加取地址上的值 `EDeref` 操作。

```
Inductive expr : Type :=
  | EConst (n: Z): expr
  | EVar (x: var_name): expr
  | EBinop (op: binop) (e1 e2: expr): expr
  | EUnop (op: unop) (e: expr): expr
  | EDeref (e: expr): expr.
```

相应的，赋值语句也可以分为两种情况。

```
Inductive com : Type :=
  | CSkip: com
  | CAsgnVar (x: var_name) (e: expr): com
  | CAsgnDeref (e1 e2: expr): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr) (c1 c2: com): com
  | CWhile (e: expr) (c: com): com.
```

# 4 更多的程序语言：WhileD

在大多数程序语言中，会同时支持或不支持取地址 `EAddrOf` 与取地址上的值 `EDeref` 两类操作，我们也可以在 WhileDeref 语言中再加入取地址操作。

```
Inductive expr : Type :=
  | EConst (n: Z): expr
  | EVar (x: var_name): expr
  | EBinop (op: binop) (e1 e2: expr): expr
  | EUnop (op: unop) (e: expr): expr
  | EDeref (e: expr): expr
  | EAddrOf (e: expr): expr.
```

程序语句的语法树不变。

```
Inductive com : Type :=
  | CSkip: com
  | CAsgnVar (x: var_name) (e: expr): com
  | CAsgnDeref (e1 e2: expr): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr) (c1 c2: com): com
  | CWhile (e: expr) (c: com): com.
```

# 5 更多的程序语言：WhileDC

下面在程序语句中增加控制流语句 continue 与 break，并增加多种循环语句。

```
Inductive expr : Type :=
  | EConst (n: Z): expr
  | EVar (x: var_name): expr
  | EBinop (op: binop) (e1 e2: expr): expr
  | EUnop (op: unop) (e: expr): expr
  | EDeref (e: expr): expr
  | EAddrOf (e: expr): expr.
```

```
Inductive com : Type :=
  | CSkip: com
  | CAsgnVar (x: var_name) (e: expr): com
  | CAsgnDeref (e1 e2: expr): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr) (c1 c2: com): com
  | CWhile (e: expr) (c: com): com
  | CFor (c1: com) (e: expr) (c2: com) (c3: com): com
  | CDoWhile (c: com) (e: expr): com
  | CContinue: com
  | CBreak: com.
```

# 6 更多的程序语言：WhileDL

下面在程序语句中增加局部变量声明。

```
C :: = skip
       V = E | * E = E |
       C; C |
       if (E) then { C } else { C } |
       while (E) do { C } |
       var V; C
```

```
Inductive com : Type :=
  | CSkip: com
  | CAsgnVar (x: var_name) (e: expr): com
  | CAsgnDeref (e1 e2: expr): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr) (c1 c2: com): com
  | CWhile (e: expr) (c: com): com
  | CLocalVar (x: var_name) (c: com): com.
```

# 7 带类型标注的表达式

下面先定义整数类型，每个整数类型由其所需比特数（`sz`）和其有无符号（`sg`）两部分构成。

```
Inductive Signedness : Type :=
  | Signed: Signedness
  | Unsigned: Signedness.
```

```
Inductive IntType : Type :=
  | Build_IntType (sz: Z) (sg: Signedness): IntType.
```

基于此，可以定义带类型标注的 While 语言表达式。

```
Inductive expr : Type :=
  | EConst (n: Z) (t: IntType): expr
  | EVar (x: var_name) (t: IntType): expr
  | EBinop (op: binop) (e1 e2: expr) (t: IntType): expr
  | EUnop (op: unop) (e: expr) (t: IntType): expr.
```

# 8 基于语法树递归定义的例子

## 8.1 表达式中包含的计算次数

下面定义 SimpleWhile 语言中一个整数类型表达式中包含的算数运算数量。

```
Fixpoint number_of_comp (ei: expr_int): Z :=
  match ei with
  | EAdd e1 e2 => number_of_comp e1 + number_of_comp e2 + 1
  | ESub e1 e2 => number_of_comp e1 + number_of_comp e2 + 1
  | EMul e1 e2 => number_of_comp e1 + number_of_comp e2 + 1
  | EVar v => 0
  | EConst n => 0
  end.
```

## 8.2 带类型表达式的几个基本合法性条件

下面我们定义两种带类型表达式的合法性条件。其一是表达式中常数的合法性条件，即每个常数都落在其类型规定的范围之内。

我们可以首先定义每个整数类型对应的数值范围。

```
Definition const_in_range (n: Z) (t: IntType): Prop :=
  match t with
  | Build_IntType sz Signed =>
      - Z.pow 2 (sz - 1) <= n < Z.pow 2 (sz - 1)
  | Build_IntType sz Unsigned =>
      0 <= n < Z.pow 2 sz
  end.
```

之后就可以定义“表达式中所有常数全都满足类型范围要求”的合法性条件。

```
Fixpoint consts_in_range (e: expr): Prop :=
  match e with
  | EConst n t =>         const_in_range n t
  | EVar v t =>           True
  | EBinop op e1 e2 t => consts_in_range e1 /\
                         consts_in_range e2
  | EUnop op e1 t =>      consts_in_range e1
  end.
```

其二是定义表达式中变量的合法性条件，即表达式中的每个变量类型都与当前环境规定的变量类型吻合。

```
Definition var_types: Type := var_name -> IntType.
```

```
Fixpoint vars_well_typed (env: var_types) (e: expr): Prop :=
  match e with
  | EConst n t =>         True
  | EVar v t =>           env v = t
  | EBinop op e1 e2 t => vars_well_typed env e1 /\
                         vars_well_typed env e2
  | EUnop op e1 t =>      vars_well_typed env e1
  end.
```