程序语言的语法

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 简单语法变换与证明

习题 1. 下面的递归函数 `remove_skip` 定义了在程序语句中删除多余空语句的操作。

```
Fixpoint remove_skip (c: com): com :=
  match c with
  | CSeq c1 c2 =>
      match remove_skip c1, remove_skip c2 with
      | CSkip, _ => remove_skip c2
      | _, CSkip => remove_skip c1
      | _, _ => CSeq (remove_skip c1) (remove_skip c2)
      end
  | CIf e c1 c2 =>
      CIf e (remove_skip c1) (remove_skip c2)
  | CWhile e c1 =>
      CWhile e (remove_skip c1)
  | _ =>
      c
  end.
```

下面请证明：`remove_skip` 之后，确实不再有多余的空语句了。所谓没有空语句，是指不出现 `c; SKIP` 或 `SKIP; c` 这样的语句。首先定义：局部不存在多余的空语句。

```
Definition not_sequencing_skip (c: com): Prop :=
  match c with
  | CSeq CSkip _ => False
  | CSeq _ CSkip => False
  | _ => True
  end.
```

其次定义语法树的所有子树中都不存在多余的空语句。

```
Fixpoint no_sequenced_skip (c: com): Prop :=
  match c with
  | CSeq c1 c2 =>
      not_sequencing_skip c /\
      no_sequenced_skip c1 /\ no_sequenced_skip c2
  | CIf e c1 c2 =>
      no_sequenced_skip c1 /\ no_sequenced_skip c2
  | CWhile e c1 =>
      no_sequenced_skip c1
  | _ =>
      True
  end.
```

下面是需要证明的结论。

```
Theorem remove_skip_no_sequenced_skip: forall c,
  no_sequenced_skip (remove_skip c).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```