1 Syntax of Commands

```
theory Com = Main:

typedecl loc — an unspecified (arbitrary) type of locations for variables
arities   loc :: term

types val   = nat — or anything else, nat used in examples
      state = loc ⇒ val
      aexp  = state ⇒ val  — arithmetic expressions are functions on states
      bexp  = state ⇒ bool — dito for boolean expressions

datatype
  com = SKIP
      | Assign loc aexp         (_ :== _  60)
      | Semi   com com          (_; _  [60, 60] 10)
      | Cond   bexp com com     (IF _ THEN _ ELSE _  60)
      | While  bexp com         (WHILE _ DO _  60)

end
```

2 Natural Semantics of Commands

```
theory Natural = Com:
```

2.1 Definition

Execution of commands

```
consts  evalc  :: (com × state × state) set
        @evalc :: [com,state,state] ⇒ bool (⟨_,_⟩/ -c→ _ [0,0,50] 50)

translations  ⟨c,s⟩ -c→ s' == (c,s,s') ∈ evalc

consts
  update :: ('a ⇒ 'b) ⇒ 'a ⇒ 'b ⇒ ('a ⇒ 'b) (_/[_/::=/_] [900,0,0] 900)


inductive evalc
intros
  Skip:    ⟨SKIP,s⟩ -c→ s
  Assign:  ⟨x :== a,s⟩ -c→ s[x::=a s]

  Semi:    ⟦ ⟨c0,s⟩ -c→ s''; ⟨c1,s''⟩ -c→ s' ⟧ ⟹ ⟨c0; c1, s⟩ -c→ s'

  IfTrue:  ⟦ b s; ⟨c0,s⟩ -c→ s' ⟧ ⟹ ⟨IF b THEN c0 ELSE c1, s⟩ -c→ s'
  IfFalse: ⟦ ¬b s; ⟨c1,s⟩ -c→ s' ⟧ ⟹ ⟨IF b THEN c0 ELSE c1, s⟩ -c→ s'
```

```
WhileFalse: ¬b s ⟹ ⟨WHILE b DO c,s⟩ -c→ s
WhileTrue:  ⟦ b s;  ⟨c,s⟩ -c→ s'';  ⟨WHILE b DO c, s''⟩ -c→ s' ⟧
             ⟹ ⟨WHILE b DO c, s⟩ -c→ s'
```

lemmas *evalc.intros [intro]* — use those rules in automatic proofs

The induction principle induced by this definition looks like this:

```
⟦⟨xc,xb⟩ -c→ xa; ⋀s. P SKIP s s; ⋀a s x. P (x :== a ) s (s[x::=a s]);
   ⋀c0 c1 s s' s''.
      ⟦⟨c0,s⟩ -c→ s''; P c0 s s''; ⟨c1,s''⟩ -c→ s'; P c1 s'' s'⟧
      ⟹ P (c0; c1) s s';
   ⋀b c0 c1 s s'.
      ⟦b s; ⟨c0,s⟩ -c→ s'; P c0 s s'⟧ ⟹ P (IF b THEN c0 ELSE c1) s s';
   ⋀b c0 c1 s s'.
      ⟦¬ b s; ⟨c1,s⟩ -c→ s'; P c1 s s'⟧ ⟹ P (IF b THEN c0 ELSE c1) s s';
   ⋀b c s. ¬ b s ⟹ P (WHILE b DO c) s s;
   ⋀b c s s' s''.
      ⟦b s; ⟨c,s⟩ -c→ s''; P c s s''; ⟨WHILE b DO c,s''⟩ -c→ s';
         P (WHILE b DO c) s'' s'⟧
      ⟹ P (WHILE b DO c) s s'⟧
⟹ P xc xb xa
```

(⋀ and ⟹ are Isabelle's meta symbols for ∀ and ⟶)

The rules of *evalc* are syntax directed, i.e. for each syntactic category there is always only one rule applicable. That means we can use the rules in both directions. The proofs for this are all the same: one direction is trivial, the other one is shown by using the *evalc* rules backwards:

lemma *skip:*
```
  (⟨SKIP,s⟩ -c→ s') = (s' = s)
  by (rule, erule evalc.elims) auto
```

lemma *assign:*
```
  (⟨x :== a,s⟩ -c→ s') = (s' = s[x::=a s])
  by (rule, erule evalc.elims) auto
```

lemma *semi:*
```
  (⟨c0; c1, s⟩ -c→ s') = (∃s''. ⟨c0,s⟩ -c→ s'' ∧ ⟨c1,s''⟩ -c→ s')
  by (rule, erule evalc.elims) auto
```

lemma *ifTrue:*
```
  b s ⟹ (⟨IF b THEN c0 ELSE c1, s⟩ -c→ s') = (⟨c0,s⟩ -c→ s')
  by (rule, erule evalc.elims) auto
```

lemma *ifFalse:*

```
  ¬b s ⟹ (⟨IF b THEN c0 ELSE c1, s⟩ -c→ s') = (⟨c1,s⟩ -c→ s')
  by (rule, erule evalc.elims) auto

lemma whileFalse:
  ¬ b s ⟹ (⟨WHILE b DO c,s⟩ -c→ s') = (s' = s)
  by (rule, erule evalc.elims) auto

lemma whileTrue:
  b s ⟹
  (⟨WHILE b DO c, s⟩ -c→ s') =
  (∃s''. ⟨c,s⟩ -c→ s'' ∧ ⟨WHILE b DO c, s''⟩ -c→ s')
  by (rule, erule evalc.elims) auto
```

Again, Isabelle may use these rules in automatic proofs:

```
lemmas evalc_cases [simp] = skip assign ifTrue ifFalse whileFalse semi whileTrue
```

2.2 Execution is deterministic

The following proof presents all the details:

```
theorem com_det: ⟨c,s⟩ -c→ t ∧ ⟨c,s⟩ -c→ u ⟶ u=t
proof clarify — transform the goal into canonical form
  assume ⟨c,s⟩ -c→ t
  thus ⋀u. ⟨c,s⟩ -c→ u ⟹ u=t
  proof (induct set: evalc)
    fix s u assume ⟨SKIP,s⟩ -c→ u
    thus u = s by simp
  next
    fix a s x u assume ⟨x :== a,s⟩ -c→ u
    thus u = s[x::=a s] by simp
  next
    fix c0 c1 s s1 s2 u
    assume IH0: ⋀u. ⟨c0,s⟩ -c→ u ⟹ u = s2
    assume IH1: ⋀u. ⟨c1,s2⟩ -c→ u ⟹ u = s1

    assume ⟨c0;c1, s⟩ -c→ u
    then obtain s' where
      c0: ⟨c0,s⟩ -c→ s' and
      c1: ⟨c1,s'⟩ -c→ u
      by auto

    from c0 IH0 have s'=s2 by blast
    with c1 IH1 show u=s1 by blast
  next
    fix b c0 c1 s s1 u
    assume IH: ⋀u. ⟨c0,s⟩ -c→ u ⟹ u = s1
```

```
    assume b s and ⟨IF b THEN c0 ELSE c1,s⟩ -c→ u
    hence ⟨c0, s⟩ -c→ u by simp
    with IH show u = s1 by blast
  next
    fix b c0 c1 s s1 u
    assume IH: ⋀u. ⟨c1,s⟩ -c→ u ⟹ u = s1

    assume ¬b s and ⟨IF b THEN c0 ELSE c1,s⟩ -c→ u
    hence ⟨c1, s⟩ -c→ u by simp
    with IH show u = s1 by blast
  next
    fix b c s u
    assume ¬b s and ⟨WHILE b DO c,s⟩ -c→ u
    thus u = s by simp
  next
    fix b c s s1 s2 u
    assume IH_c: ⋀u. ⟨c,s⟩ -c→ u ⟹ u = s2
    assume IH_w: ⋀u. ⟨WHILE b DO c,s2⟩ -c→ u ⟹ u = s1

    assume b s and ⟨WHILE b DO c,s⟩ -c→ u
    then obtain s' where
      c: ⟨c,s⟩ -c→ s' and
      w: ⟨WHILE b DO c,s'⟩ -c→ u
      by auto

    from c IH_c have s' = s2 by blast
    with w IH_w show u = s1 by blast
  qed
qed
```

This is the proof as it is presented in the lecture. The remaining cases are simple enough to be proved automatically:

```
theorem ⟨c,s⟩ -c→ t ∧ ⟨c,s⟩ -c→ u ⟶ u=t
proof clarify
  assume ⟨c,s⟩ -c→ t
  thus ⋀u. ⟨c,s⟩ -c→ u ⟹ u=t
  proof (induct set: evalc)
    fix s u assume ⟨SKIP,s⟩ -c→ u
    thus u = s by simp
  next
    fix b c s s1 s2 u
    assume IH_c: ⋀u. ⟨c,s⟩ -c→ u ⟹ u = s2
    assume IH_w: ⋀u. ⟨WHILE b DO c,s2⟩ -c→ u ⟹ u = s1

    assume b s and ⟨WHILE b DO c,s⟩ -c→ u
    then obtain s' where
```

```
      c: ⟨c,s⟩ -c→ s' and
      w: ⟨WHILE b DO c,s'⟩ -c→ u
      by auto

    from c IH_c have s' = s2 by blast
    with w IH_w show u = s1 by blast
  qed (best dest: evalc_cases [THEN iffD1])+ — prove the rest automatically
qed

end
```