Grundlagen des Software Engineering
Fundamentals of Software Engineering

Winter Term 2011/12

Prof. Dr. Dr. h. c. Dieter Rombach

Chapter 6.2: Software Unit Engineering - Functional Specification
The goals of this chapter are to

- understand how to systematically develop code from specs (using functional semantics)
- Understand the benefits of stepwise refinement based on level-completeness (also how to benefit from the rules when programming informally)
- Understand the benefits of stepwise verification based on level-completeness

Literature

Why are functional semantics natural (by example)?

- **Specs**

```plaintext
procedure SORT (var L: list of integers)
(L ≠ Ø) → {(L, L'): L' = permutation(L) and 
∀i with 1 ≤ i < size(L'): L'[i] ≤ L'[i + 1]) } | () }
```

- **Program execution (e.g., debugging)**
  - States
  - State transitions via program statement execution
Functional specs (general form)

- **F**: Input $\rightarrow$ Output has the form of a **conditional assignment**:
  
  $$(\text{Cond}_1 \rightarrow f_1 \mid \text{Cond}_2 \rightarrow f_2 \mid \ldots \mid \text{Cond}_n \rightarrow f_n), \text{ where}$$

  - $\text{Cond}_i$ are Boolean expressions about the set of input variables.
  - $f_i$ define the set of all logical in-/output pairs (partial function)
  - $f_i$ can be described by
    
    $\rightarrow$ listing all pairs as set $[f_i: \{(1,1), (2,4), \ldots\}]$
    
    $\rightarrow$ analytical representation $[f_i(x) = x^2]$
    
    $\rightarrow$ informal text
    
    $\rightarrow$ parallel assignments

  - this form of assignment can be applied in parallel to several variables:
    
    $$(x_1, x_2, \ldots, x_n) := (g_1(x_1, \ldots, x_n), \ldots, g_n(x_1, \ldots, x_n))$$

  - this form of assignment defines pre-/post assertions

  - **Example**: procedure example (var A, B: integer);
    
    $$(A \leq B \rightarrow A, B := (B - (B - A) / 2), (B - (B - A) / 2)) | ()$$

  - Each $f_i$ can itself be a conditional instruction.
    
    $\rightarrow$ Example:
    
    $$(x \geq 0 \rightarrow (y \geq 0 \rightarrow (x, y) := (y, x) | () ) | () )$$

    corresponds to
    
    $$(x \geq 0 \land y \geq 0 \rightarrow (x, y) := (y, x) | () )$$

- **Divide & Conquer at Spec. Level!**
**Introduction**

- **Functional Semantics**

- **Stepwise Refinement**

- **Verification**

---

**Functional specs (example)**

- \((^* f: (\text{input} = (i_1 | i_2 | i_3 | ... | i_n) \text{ with } n \geq 2 \land i_1 \leq i_2 \rightarrow \text{output} := (("X: ", i_2 - (i_2 - i_1) / 2), ("Y: ", i_2 - (i_2 - i_1) / 2)) | \text{output} := (("X: ", i_1), ("Y: ", i_2))*)\)

---

**program xy (input, output)**

```
var X: integer;
    Y: integer;

procedure example (var A, B: integer);
(* h: (A \leq B \rightarrow A, B := (B - (B - A) / 2), (B - (B - A) / 2) ) | () *)
    while A < B do
        begin
            A := A + 1;
            if A < B then
                B := B - 1;
        end;
    end;
end;
```

**Is this a correct implementation?**

```
begin (* xy *)
    read(X);
    read(Y);
    example(X, Y);
    writeln(‘X: ‘, X);
    writeln(‘Y: ‘, Y);
end. (* xy *)
```
Functional semantics (example)

(* f *)

program xy (input, output)
   var X: integer;
       Y: integer;
   procedure example (var A, B: integer);
      h: (A ≤ B → A, B := (B - (B - A) / 2), (B - (B - A) / 2) ) | ()
   end; (* example *)
Functional semantics (example)

(* f *)

program xy (input, output)
  var X: integer;
  Y: integer;

  procedure example (var A, B: integer);
  (* h: (A ≤ B → A, B := (B - (B - A) / 2), (B - (B - A) / 2) ) | () *)
    while A < B do
      h1: (A + 1 < B → A, B := A + 1, B - 1) | (A := A + 1)
    end;
  end;
end; (* example *)
Functional semantics (example)

(* f *)

program xy (input, output)

    var X: integer;
    Y: integer;

    procedure example (var A, B: integer);
    (* h: (A ≤ B → A, B := (B - (B - A) / 2), (B - (B - A) / 2)) | () *)
    while A < B do
        (* h₁: (A + 1 < B → A, B := A + 1, B - 1) | (A := A + 1) *)
        begin
            h₁₁: A := A + 1 *;
            h₁₂: (A < B → B := B - 1) | ()
        end;
    end;
end; (* example *)
Functional semantics (example)

(* f *)

program xy (input, output)
  var X: integer;
      Y: integer;

procedure example (var A, B: integer);
  (* h: (A ≤ B → A, B := (B - (B - A) / 2), (B - (B - A) / 2) ) | () *)
  while A < B do
    (* h₁: (A + 1 < B → A, B := A + 1, B - 1) | (A := A + 1) *)
    begin
      (* h₁₁: A := A + 1 *)
      A := A + 1;
      h₁₂: (A < B → B := B - 1) | ()
      end;
  end;
end; (* example *)
Functional semantics (example)

```pascal
(* f *)
program xy (input, output)
  var X: integer;
  Y: integer;

  procedure example (var A, B: integer);
  (* h: (A ≤ B → A, B := (B - (B - A) / 2), (B - (B - A) / 2) ) | () *)
  while A < B do
    (* h₁: (A + 1 < B → A, B := A + 1, B - 1) | (A := A + 1) *)
    begin
      (* h₁₁: A := A + 1 *)
      A := A + 1;
      (* h₁₂: (A < B → B := B - 1) | () *)
      if A < B then
        h₁₂₁: B := B - 1
      end;
    end;
  end; (* example *)
```
Functional semantics (example)

(* f *)

program xy (input, output)

var X: integer;
    Y: integer;

procedure example (var A, B: integer);

(* h: (A ≤ B → A, B := (B - (B - A) / 2), (B - (B - A) / 2)) | () *)

while A < B do

(* h₁: (A + 1 < B → A, B := A + 1, B - 1) | (A := A + 1) *)

begin

(* h₁₁: A := A + 1 *)
A := A + 1;

(* h₁₂: (A < B → B := B - 1) | () *)

if A < B then

(* h₁₂₁: B := B - 1 *)
B := B - 1;

end;

end;

end; (* example *)

Stepwise verification would be easier!

How do we get these annotations?

- A priori (design patterns)
- A posteriori (reading/verification)
A program state $S$ is a function $S: \text{var} \rightarrow \text{val}$, where
- $\text{var}$ is a set of program variables and
- $\text{val}$ is a set of values.

$\Rightarrow$ A program state can be represented by the set of all pairs $(x \in \text{var}, a \in \text{val})$.

$[\text{pgm}]$ is the function that describes the functional behaviour of the program part ‘pgm’.

$[\text{program X (input, output); decl; } S_1; S_2; \ldots; S_n \text{ end.}] = [\text{end.}] \circ [S_n] \circ \ldots \circ [S_1] \circ [\text{decl}] \circ [\text{program X (input, output)}]$
Functional semantics

- We describe the behaviour of whole programs through
  - the semantics of elementary program parts
  - the set theory operation

- **[program X (input, output)] : input \rightarrow state**

  \[\text{[program X (input, output)] := \{(input, s): s = \{(input, _...), (output, $)\}\}}\]

- **[end. ] : state \rightarrow output**

  \[\text{[end.] := \{(s, output): s = \{..., (output, xxx_...), ...\}, output = xxx\}}\]

- **[expr] : state \rightarrow val**

  \[\text{[x + y] := \{(s, c): s = \{(x, a), (y, b), ...\}, c = [x](s) + [y](s) = a + b\}}\]
Software UE – Functional Specification

Functional semantics

- [Decl] : state → state
  [var x: integer] := {(s, t): s = t, except that, in addition, t contains a pair (x, ?)}

- [Statement] : state → state
  [z := x + y] := {(s, t): s = t, except that [z](t) = [x](s) + [y](s)}

- Sequence: [S1; S2] : state → state
  [S1, S2] := [S2] o [S1]
  [begin S1 end] := [S1]

- Alternative: [if B then S1 else S2] : state → state
  [if B then S1 else S2] := {(s, [S1](s)): [B](s) = true} ∪ {(s, [S2](s)): [B](s) = false}

- Iteration: [while B do S] : state → state
  [while B do S] := [if B then begin S; while B do S end]
Functional semantics

- Procedure / Function Calls
- Example

```pascal
program xy (input, output)
...
procedure example (var A, B: integer);
(* A ≤ B → A, B := (B - (B - A) / 2), (B - (B - A) / 2) *)
[T]
begin (* xy *)
  read(X);
  read(Y);
  example(X,Y);
  writeln('X: ', X);
  writeln('Y: ', Y);
end. (* xy *)
```
Functional semantics

- **Procedure / Function Calls**
- **Definition**

\[
\begin{align*}
\text{example}(X=A,=B=Y) & := [T] \\
& \left\{ 
(X, Y) \\
(A, B)
\right. \\
\end{align*}
\]

- **Application**

\[
\begin{align*}
\text{example}(X,Y) & (..., X=2, Y=8, ...) = \\
(X \leq Y & \rightarrow X, Y := (Y-(Y-X)/2), (Y-(Y-X)/2)) \mid () (..., X=2, Y=8, ...) = (5, 5)
\end{align*}
\]
In each elementary design task, the issue is to select the appropriate refinement operator \( \text{op} \in \{\text{seq, alt, it}\} \) as well as the \( f_1, \ldots, f_n \).

We distinguish (depending on the form of \( f \) and the type of operator) the following situations:

<table>
<thead>
<tr>
<th>Operators</th>
<th>Form of ( f )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Simultaneous Statement</td>
</tr>
<tr>
<td>Sequence</td>
<td>(1)</td>
</tr>
<tr>
<td>Alternative</td>
<td>(8)*</td>
</tr>
<tr>
<td>Iteration</td>
<td>(9)*</td>
</tr>
<tr>
<td>Recursion</td>
<td>(10)*</td>
</tr>
</tbody>
</table>
- All pairs (spec, impl) are possible!
- (1) ... (5) represents natural pairs (spec, impl). For this, we want to give design rules / heuristics!

- (5)** ... (7)** represent unnatural pairs to the extent that in these cases, specification is unnecessarily complicated. In these cases, we give no design rules / heuristics! However, such implementations can be proven as correct!

  ```
  /* f: (x \geq y \rightarrow (x, y = y, x) | (x, y = y, x)) */
  temp = x;
  x = y;
  y = temp;
  ```

- (8)* ... (11)* represent unnatural pairs to the extent that in these cases implementations is unnecessarily complicated. In these cases, we give no design rules / heuristics! However, such implementations can be proven as correct!

  ```
  /* f: (x, y = y, x) */
  if (x <= y)
  {
    temp = x;
    x = y;
    y = temp;
  }
  else
  {
    temp = x;
    x = y;
    y = temp;
  }
  ```
Design Heuristics
Situation (1)

- Data

\[ f = (x_1, x_2, ..., x_n = y_1, y_2, ..., y_n) \]

\[ \text{op} = \text{seq} \ (\text{d.h. P: (S}_1; S_2; ...; S_n)) \]

- Rules

  - **R(1):** All original values of \( y_1 \) must be preserved in each step.

  - **R(2):** In each step: Assign \( x_i \) their intended values, as long as this does not interfere with R(1). If no assignments are possible, create a temporary variable and assign it a value to be preserved.

  - **R(3):** Stop if all \( x_i \) have received their intended values.

- Example

\[
\begin{align*}
/* f: (x, y = y, x) */ \\
x &= y; \quad \text{Contradiction to R(1)} \\
\text{temp} &= x; \\
x &= y; \\
y &= \text{temp};
\end{align*}
\]

- Note

  - Keeping this rule guarantees correctness [and possibly makes correctness proof unnecessary]
**Design Heuristics**

### Situation (2)

#### Data

\[ f = (B_1 \rightarrow C_1 \mid ... \mid B_n \rightarrow C_n) \quad /*\text{conditional statement}*/ \]

\[ \text{op = alt (i.e., P: if (B) S)} \]

#### Transformation Rule R(1)

```plaintext
if B_1
  { C_1 }
else if B_2
  { C_2 }
else if B_n
  { C_n }
```

#### Optimization

- Leave out those cases whose conditions can never become true!
  - (z.B. \( a \leq b \rightarrow s_1 \mid a > b \rightarrow s_2 \mid s_n \))

- Combine (i.e., simplify) nested conditions!
  - (z.B. \( a \leq b \rightarrow ((a \geq b) \rightarrow s_1)) \leftrightarrow ((a = b) \rightarrow s_1))\)
Design Heuristics
Situation (3)

Data

\[ f = (B_1 \rightarrow C_1 | ... | B_n \rightarrow C_n) \] /* conditional statement */

\[ \text{op} = \text{it} \text{ (i.e., P: while (B) S)} \]

Rules (derived from verification rules)

- There is no implementation of \( f \) through \text{while}-constructs, if one of the following rules is violated (derived from verification rules 1 and 2)
  \[ \rightarrow R(1a): \quad \text{picture (f)} \subseteq \text{Def (f)} \]
  \[ \rightarrow R(1b): \quad \forall x \in \text{picture (f)} : f(x) = x \]

- Determine B \( \text{in such a way that the following is true:} \)
  \[ \rightarrow H(2a): \quad [B] = \text{true, for all values in (Def(f) - picture(f))} \]
  \[ \rightarrow H(2b): \quad [B] = \text{false, for all values in picture(f)} \]

- Determine S \( \text{in such a way that the following is true:} \)
  \[ \rightarrow H(3a): \quad [\text{if (B) S}] \text{ gets all values that } f \text{ needs for defining the final state.} \]
  \[ \rightarrow H(3b): \quad \text{while (B) S terminates for all states } \in \text{Def(f)} \]
**Design Heuristics**

**Situation (3)**

- **Example 1** \( f = \text{Id} \)
  - (1) Rule (1a) and (1b)
  - (2) \( B = \text{false} \) is one possibility (or \( x \neq x \))
  - (3) At will! (z.B.: \( x = x + 1 \))

- **Example 2** \( f : x = y \)
  - (1) \( \checkmark \)
  - (2) \( B = x < y \)
  - (3) \( x = x + 1 \)

There is no implementation of \( f \) through **while**-constructs, if one of the following rules is violated (derived from verification rules 1 and 2)

\[
\begin{align*}
R(1a): & \quad \text{picture (f)} \subseteq \text{Def (f)} \\
R(1b): & \quad \forall x \in \text{picture (f)} : f(x) = x \\
\end{align*}
\]

Determine \( B \) in such a way that the following is true:

\[
\begin{align*}
H(2a): & \quad [B] = \text{true, for all values in (Def(f) - picture(f))} \\
H(2b): & \quad [B] = \text{false, for all values in picture(f)} \\
\end{align*}
\]

Determine \( S \) in such a way that the following is true:

\[
\begin{align*}
H(3a): & \quad \text{if (B) S gets all values that f needs for defining the final state.} \\
H(3b): & \quad \text{while (B) S terminates for all states } \\
& \quad \text{in Def(f)} \\
\end{align*}
\]

---

**Homework!**
Design Heuristics

Situation (4)

- Data

\[ f(p) = (B(p) \rightarrow f(K(p) \mid H(p))) \quad /*\text{repetitive recursive function}*/ \]

op = it  (i.e.,  P: \textbf{while} (B) \{S\})

- Note

  - (0) p: List of formal parameters
  - (1) K(p) is the function that calculates arguments for recursive call
  - (2) H(p) is the function that calculates results

- Transformation Rule R(1) for Repetitive Recursions

```c
<Typ> f (-p);
/* (v = p) */
{
    \textbf{while} (B(v))
    {
        /* v = K(v) */
        \}
    return H(v);
}
```
Example

\[ \text{mod: } ((a \geq b) \rightarrow \text{mod (a-b, b) | a}) \]

\[ f = \text{mod} \]

\[ P = a, b \text{ (int)} \]

\[ B = a \geq b \]

\[ K = a - b, b \]

\[ H = a \]

\[
\begin{align*}
\text{int mod (int a, int b)} \\
\{} \\
\text{int c, d} \\
/\ast (c, d = a, b) */ \\
c = a; \\
d = b; \\
\text{while (c \ d)} \\
/\ast(c, d = c - d, d) */ \\
c = c - d; \\
\text{return c;}
\end{align*}
\]
For the verification of sequence and alternative use functional construction operators defined earlier

For the verification of loops, the following three criteria must be checked (without proof, see Zelkowitz (IEEE Computer, Nov. 90, 30-39)):

- \( \text{domain}(f) = \text{domain}([\text{while } B \text{ do } S]) \)
- \( (\neg(B) \Rightarrow f) = (\neg(B) \Rightarrow ()) \)
- \( f = [\text{if } B \text{ then } S] \circ f \)
Formal Verification

- **Units of Type I:**
  - According to functional semantics, two steps are necessary for this:
    - (1) **Determine actual meaning function of program** \([Pgm]\)
      - **formally:** application of the concatenation rules / symbolical execution (see: later)
      - Or
      - **informally:** Bottom-up abstraction through stepwise abstraction reading of the code
    - (2) **Prove for intended meaning function (= requirements)** \(f : f \leq [Pgm]\)

- **Example - Body of the procedure ‘example’**

```plaintext
(* f : (A ≤ B → A, B := (B - (B - A) / 2), (B - (B - A) / 2) ) ) | () *)
while A < B do
begin
A := A + 1;
if A < B then
  B := B - 1;
end;
```
Example - Body of the procedure ‘example’

**Prove**

\[
\begin{align*}
\text{[L3]} & : \quad \text{A:=A+1} \\
& \quad \quad \quad \text{(Axiom)} \\
\text{[L5]} & : \quad \text{B:=B-1} \\
& \quad \quad \quad \text{(Axiom)} \\
\text{[L4 ... L5]} & : \quad (A<B \rightarrow B:=B-1) \ | \ () \\
\text{[L2 ... L6]} & : \quad (A+1<B \rightarrow A:=A+1, B-1) \ | \ (A:=A+1) \\
& \quad \quad \quad \text{(two execution tables)} \\
\text{[L1 ... L6]} & : \quad (A\leq B \rightarrow A, B:=(B-(B-A)/2), (B-(B-A)/2)) \ | \ () \ \\
& \quad \quad \quad \text{(application of the verification rules)}
\end{align*}
\]

\[
\begin{align*}
\text{(* f : (A \leq B \rightarrow A, B := (B - (B - A) / 2),} \\
\quad (B - (B - A) / 2) ) | () *)
\end{align*}
\]
Introduction

Functional Semantics

Stepwise Refinement

Verification

Formal Verification

program xy (input, output)

var X: integer;
    Y: integer;

procedure example (var A, B: integer);

(* h: (A ≤ B → A, B := (B - (B - A) / 2), (B - (B- A) / 2) ) | () *)

while A < B do

(* h₁: (A + 1 < B → A, B := A + 1, B - 1) | (A := A + 1) *)

begin

(* h₁₁: A := A + 1 *)

A := A + 1;

(* h₁₂: (A < B → B := B - 1) | () *)

if A < B then

(* h₁₂₁: B := B - 1 *)

B := B - 1;

end;

end;

end; (* example *)

Stepwise abstraction reading provides meaningful & consistent documentation a posteriori!

Easy to understand incrementally!
Easy to change (traceability)
Reading by Stepwise abstraction

\[
\text{if } x \neq 0 \text{ then}
\begin{align*}
  & y := 5 \\
\text{else} & \quad (x \neq 0 \rightarrow y := 5 \mid z := z - x)
\end{align*}
\]

\[
\text{if } z > 1 \text{ then}
\begin{align*}
  & z := z \times x \\
\text{else} & \quad (z > 1 \rightarrow z := z \times x \mid z := 0)
\end{align*}
\]

\[
\text{else}
\begin{align*}
  & z := 0
\end{align*}
\]

\[\text{if } x \neq 0 \rightarrow (z > 1 \rightarrow y, z := 5, z \times x \mid y, z := 5,0) \mid
\text{(z > 1 \rightarrow z := (z - x) \times x \mid z := 0))}\]

Clear documentation

- [line 2]: \( y := 5 \)
- [line 4]: \( z := z - x \)
- [line 6]: \( z := z / x \)
- [line 8]: \( z := 0 \)
- [line 1 ... 4]: \((x \neq 0 \rightarrow y := 5 \mid z := z - x)\)
- [line 5 ... 8]: \((z > 1 \rightarrow z := z / x \mid z := 0)\)
- [line 1 ... 8]: \((x \neq 0 \rightarrow (z > 1 \rightarrow y, z := 5, z / x \mid y, z := 5,0) \mid
(z > 1 \rightarrow z := (z - x) / x \mid z := 0))\)

Abstraction functions could also be specified informally!
Reading by Stepwise abstraction

(1) Understanding:
Abstract the actual meaning (via stepwise abstraction reading) \([\text{pgm}]\) from the code.

(2) Discovery of possible failure(s):
Compare the abstracted actual meaning function with the given specification \(f\) in order to diagnose possible failures.

\[ f \subseteq [\text{pgm}] \]

Note: If specifications of individual design parts exist, then the diagnosis can be made step by step.

(3) Isolating faults
Look for the cause of failures discovered (i.e., faults) in code.
Note: Understanding the code by reading first should facilitate localization.

(4) [Corrections]

Documents used

<table>
<thead>
<tr>
<th>Steps</th>
<th>Specification (f)</th>
<th>Source Code ([\text{pgm}])</th>
<th>Executable unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discovery of failures</td>
<td>Reading (\rightarrow [\text{pgm}])</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Comparison (\rightarrow f \subseteq [\text{pgm}])</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Isolating faults</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Symbolic execution allows the simple (mechanical) calculation of concatenated program parts (sequence, alternative) via so-called **Execution Tables**

Execution Tables use variable names instead of variable values for execution \(\Rightarrow\) verification activity!

**(l) Sequence:**
- Example: \([\text{temp} := x; x := y; y := \text{temp}] := [y := \text{temp}] \circ [x := y] \circ [\text{temp} := x]\)

<table>
<thead>
<tr>
<th></th>
<th>(x)</th>
<th>(y)</th>
<th>(\text{temp})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{temp} := x)</td>
<td></td>
<td></td>
<td>(x)</td>
</tr>
<tr>
<td>(x := y)</td>
<td>(y)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(y := \text{temp})</td>
<td></td>
<td></td>
<td>(x)</td>
</tr>
</tbody>
</table>

\((x, y, \text{temp}) := (y, x, x)\)
**Symbolic Execution**

- **(II) Alternative:**
  - **Example:** \([x := x + y; y := y - x; \text{if } x + y > 0 \text{ then } y := x + y \text{ else } y := -x - y]\)

<table>
<thead>
<tr>
<th>Condition</th>
<th>(x)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x := x + y)</td>
<td>(x + y)</td>
<td>(x + y)</td>
</tr>
<tr>
<td>(y := y - x)</td>
<td>(-x)</td>
<td>(-x)</td>
</tr>
<tr>
<td>if (x + y &gt; 0)</td>
<td>(y &gt; 0 \text{ (true)})</td>
<td>(y &gt; 0 \text{ (true)})</td>
</tr>
<tr>
<td>(y := x + y)</td>
<td>(y)</td>
<td>(y)</td>
</tr>
</tbody>
</table>

(i) \((y > 0 \rightarrow x, y := x + y, y)\)

<table>
<thead>
<tr>
<th>Condition</th>
<th>(x)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x := x + y)</td>
<td>(x + y)</td>
<td>(x + y)</td>
</tr>
<tr>
<td>(y := y - x)</td>
<td>(-x)</td>
<td>(-x)</td>
</tr>
<tr>
<td>if (x + y &gt; 0)</td>
<td>(y &gt; 0 \text{ (true)})</td>
<td>(y &gt; 0 \text{ (true)})</td>
</tr>
<tr>
<td>(y := x + y)</td>
<td>(y)</td>
<td>(y)</td>
</tr>
</tbody>
</table>

(ii) \((y \leq 0 \rightarrow x, y := x + y, -y)\)

(i) \& (ii) \((y > 0 \rightarrow x, y := x + y, y \mid x, y := x + y, -y)\)