

### Semantics

- A programming language specification consists of a syntactic description and a semantic description.
- Syntactic description:symbols we can use in a language
- **Semantic Description**: what phrases in a programming language mean.
- Semantics may be given as
- Denotational
- Axiomatic
- Operational
- Here we concentrate on axiomatic descriptions: the meaning is defined by a logical calculus called **program logic** which provides a tool for the derivation of programs and assertions of the form  $\{Q\} p$ {R}

### Example:

• Read in 2 integers and output their product divided by their sum. You are guaranteed that there are 2 integer values in the input stream.

- Precondition = { Input stream contains two integer values }
- Postcondition = { Product / Sum is output }

 $\{Q\} p \{R\}$  states that a program p. once started in a state satisfying  $\{Q\}$  will lead to a situation characterised by  $\{R\}$ 

- $\{Q\}$  may also be written as the weakest precondition of p tp achieve postcondition R i.e. wp(p, R)
- Wp(S, R) represents the set of all states such that execution of S beginning in any one of them is guaranteed to terminate in a finite amount of time satisfying R.
- Examples:
- wp (i = i +1, i <= 1)
- S: if x>=y then z = x else z = y, R: z = max(x,y) calculate wp(S,R)
- Let S be as above and R: z=y, calculate wp(S,R)
- S: if x>=y then z = x else z = y,R: z = y-1
- calculate wp(S,R)
- Let S be as above, R: z=y+1

- Command S is usually designed to establish the truth of a postcondition R. We may not be interested in wp(S,R). If we can find a stronger precondition Q that represents a subset of the set wp(S,R) and can show Q => wp(S,R) then we are content with Q as the postcondition.
- When we write {Q} p {R} we denote Total Correctness
  Q {p} R denotes partial correctness.

## Some properties of wp

- Law of excluded miracle: wp(S,F) = F
- Distributivity of conjunction:  $wp(S,Q) \land wp(S,R) = wp(S,Q \land R)$
- Law of monotonicity: if Q => R then wp(S,Q) => wp(S,R)
- Distributivity of disjunction: wp(S,Q) ∨wp(S,R) => wp(S,Q ∨ R)

#### Nondeterministic:

- Execution of a command is nondeterministic if it need not always be exactly the same each time it is begun in the same state
- e.g.  $\{x = 4\} \ x := 14 \parallel x := x+1 \ \{?\}$

### Exercises

#### Determine

- wp (i:= i +1, i >0)
- wp(i = i + 2; j = j 2, i + j = 0)
- $\begin{array}{l} \ wp(i=i+1;j=j\ \text{-}1,\ i\ *j=0) \\ \ wp(z=z^*j;\ i\ :=i\ \text{-}1,\ z\ *\ j^i=c) \end{array}$
- wp(a[i] = 1, a[i] = a[j])
- wp(a[a[i]] = i, a[i]=i)

## Skip & Abort

#### Skip

- Execution of the skip command does nothing.
- It is equivalent to the empty command;
- It is the identity transformer
- wp(skip, R) = R

#### Abort

- wp(abort, R) = False
- Abort should never be executed as it may only be executed in a state satisfying False.

# Sequential Composition

- A way of composing larger programs from smaller segments
- If s1 and S2 are commands then s1;s2 is a new command
- wp(s1;s2, R) = wp(s1, wp(s2, R))

### Assignment

- x := e
- x is a simple variable, e is an expression and the types of x and e are the same
- $wp(x := e, R) = domain(e) cand R_e^{x}$
- Domain(e) is a predicate that describes the set of all states in which e may be evaluated i.e. is well defined.
- Usually we write:  $wp(x := e, R) = R_e^x$

### **Examples:**

- wp(x:=5, x =5)
- wp(x:=5, x !=5)
- wp(x:=x+1, x < 10)
- wp(x:=  $x^*x$ ,  $x^4 = 10$ )
- wp(x:=a/b, p(x))
- wp(x:=b[i], x=b[i]) for b, an array with indexes 0 ..100

# Multiple Assignment

#### Multiple assignment has the form

x1, x2, x3, ..., xn := e1, e2, e3, ..., en where xi are distinct simple variables and ei are expressions.

#### Definition:

$$\begin{split} & wp(x1, x2, x3, \ldots, xn := e1, e2, e3, \ldots, en, R) \\ & = domain(e1, e2, e3, \ldots, en) \ cand \ R_{e1, e2, e3, \ldots, en} \, ^{x1, \, x2, \, x3, \, \ldots, \, xn} \end{split}$$

Examples:  $\begin{array}{ll} x,y:=y,x;\\ x,y,z:=y,z,x\\ wp(z,y{:=}z^*x,\,y{\text{-}}1,\,y{\text{>=}}0\wedge z^*x^y=c) \end{array}$ 

- Execution of an expression may change only the variables indicated and evaluation of an expression may change no variables.
- This prohibits functions with side effects and allows us to consider expressions as conventional mathematical entities I.e. we can use associativity, commutativity of addition etc.
- · Example: Swapping two variables:
- wp(t:=x; x:=y; y:=t,  $x = X \land y = Y$ }

### The if statement

If  $B_1 \rightarrow S_1$ []  $B_2 \rightarrow S_2$ ... []  $B_n \rightarrow S_n$ 

fi

- Each  $B_i \rightarrow S_i$  is a guarded command and each  $S_i$  may be any command e.g. skip, about, sequential composition etc.
- If any guard B<sub>i</sub> is not well defined in the state in which execution begins, abortion may occur. This is because nothing is assumed by the order of evaluation of the guards.
- · At least one guard must be true to avoid abortion.
- If at least one guard  $B_i$  is true, then 1 guarded command  $B_i {\rightarrow} S_i$  is chosen and  $S_i$  is executed.

# Wp (If, R)

Wp(If, R) =

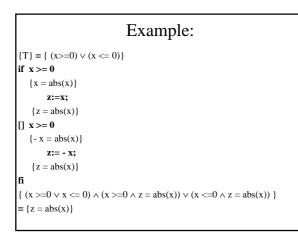
 $domain(\mathbf{BB}) \land \mathbf{BB} \land (\mathbf{B}_1 \Rightarrow wp(\mathbf{S}_1, \mathbf{R})) \land \dots \land (\mathbf{B}_n \Rightarrow wp(\mathbf{S}_n, \mathbf{R}))$ where  $\mathbf{BB} = \mathbf{B}_1 \lor \mathbf{B}_2 \lor \dots \lor \mathbf{B}_n$ 

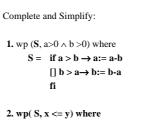
 $wp(If, R) = (\exists i : 1 \le i \le n : B_i) \land (\forall i : 1 \le i \le n : B_i \Rightarrow wp(S_i, R))$ 

**Exercises:** 

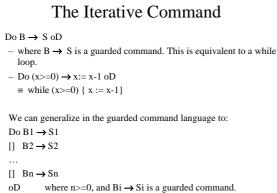
Example:  $A \equiv (if \ x \ge 0 \rightarrow z := x)$ []  $x \le 0 \rightarrow z := -x$ )

wp(A, z = abs(x)) = True





S = if  $x > y \rightarrow x, y := y, x$ []  $x <= y \rightarrow skip$ fi



Note: Non Determinism is allowed.

- Let  $BB = B1 \lor B2 \lor \ldots \lor Bn$
- $H_0(R) = \neg BB \land R$
- Represents the set of states in which execution of DO terminates in 0 iterations with R true, as the guards are initially false

```
wp(DO,R) = \exists k: 0 \le k: H_k(R)
```

 Represents the set of states in which execution of DO terminates in a bounded number of iterations with R true.

```
Example: What does the following calculate? How can we prove it?
i,s = 1, b[0];
```

**Do**  $i <> 11 \rightarrow i, s := i+1, s + b[i]$  **OD** {**R**:  $s = \Sigma k: 0 <= k < 11:b[k]$ )

# **Invariant {P}** : Predicate that is true throughout the program **Guard Bi, BB:**

- True on entry into the loop
- May be true or false at the exit point of the loop => re-evaluate guard
- The guard is always false after the loop terminates
- Postcondition {R}: The postcondition should imply the Invariant and the negation of the guard i.e.  $P \land \neg BB => R$
- **Precondition{Q}:** Should imply the Invariant with initialisations.

# Loop Template $\{Q\} \Rightarrow \{P\}$ Do BB $\{P \land BB\}$ "Loop Body" $\{P\}$ Od $\{P \land \neg BB\} \Rightarrow \{R\}$

# **Program Verification**

• Given a precondition, a postcondition and some code verify that the code when executed in a state satisfying the given precondition achieves the given postcondition.

 $\begin{array}{l} \{Q\}: \{ \mbox{Array b has values} \} \\ i,s:=1,b[0] \\ \mbox{Do $i <> N$} \\ i,s:=i+1, s+b[i]; \\ \mbox{Od} \end{array}$ 

 $\{R\}: \{s = \Sigma k: 0 \le k \le 11:b[k])\}$ 

# Loop Termination

- To show that a loop terminates we introduce an integer function, t. where t is a function of the program variables i.e. an upper bound on the number of iterations still to be performed.
- t is called the **variant function** and it is a measure of the amount of work yet to be completed by the loop.
- Each iteration of the loop decreases **t** by at least one
- As long as execution of the loop has not terminated then **t** is bounded below by 0. Hence the loop must terminate.
- In our last example t: 11-i

# Checklist for loops

- Show that **P** is true before the execution of a loop begins
- Show that  $\mathbf{P} \land \neg \mathbf{BB} \Rightarrow \mathbf{R}$  i.e. when the loop terminates the desired result is true.
- Show that {P ∧ Bi} Si {P} 1<=i<=n i.e. execution of each guarded command terminates with P true so that P is an invariant of the loop.
- Show that  $P \land BB \Rightarrow (t > 0)$  so that the bound function i.e. "the amount of work yet to be done" is bounded from below as long as the loop has not terminated.
- Show that {P ∧ Bi} t1 :=t;Si; {t<t1} for 1 <=i<=n so that each loop iteration is guaranteed to decrease the bound function. In general t can only provide an upper bound on the number of iterations to be performed. Hence, it is called the bound function or the variant function.</li>