# Denotational Semantics of Programs

**Slide 1**

**Denotational Semantics of Programs**

**Denotational Semantics of *SimpleExp***

We will define the **denotational semantics** of simple expressions using a function

$$[\![\_]\!] : SimpleExp \to \mathbb{N}.$$

**Denotational Semantics of *While***

We will later discuss the *denotational semantics* of our while programs.

### Denotational Semantics for Simple Expressions

As we have seen, operational semantics talks about how an expression is evaluated to an answer. Denotational semantics, on the other hand, has grander aspirations. The denotational semantics of a language (such as *While* and *SimpleExp*) attempts to describe what a piece of program text 'really means'. In the case of simple expressions, a piece of program text 'is really' a number, so we will define a function $[\![\_]\!]$ such that, for any expression $E$, $[\![E]\!]$ is a number, giving the meaning of $E$. Therefore, $[\![\_]\!]$ will be a function from expressions to numbers, and we write

$$[\![\_]\!] : SimpleExp \to \mathbb{N}$$

where $\mathbb{N}$ is a set of natural numbers. Given this function, the set $\mathbb{N}$ is called the *semantic domain* of *SimpleExp*, which just means it is the place where the meanings live. As we come to study more complex languages, we will find that we need more complex semantic domains. The construction and study of such domains is the subject of *domain theory*, an elegant mathematical

theory which provides a foundation for denotational semantics; unfortunately, domain theory is beyond the scope of this course.
For now, notice that our choice of semantic domain has certain consequences for the semantics of our language: it implies that every expression will 'mean' exactly one number, so without even seeing the definition of $[\![\_]\!]$, someone looking at our semantics already knows that the language is (expected to be) normalizing and deterministic.
It is easy to give the meaning to expressions that are numbers:

$$[\![n]\!] = n.$$

For expressions $E_1 + E_2$, the meaning will of course be the sum of the meanings of $E_1$ and $E_2$:

$$[\![E_1 + E_2]\!] = [\![E_1]\!] + [\![E_2]\!].$$

We could make similar definitions for multiplication and so on.

**Slide 2**

## Denotational Semantics of Simple Expressions

We define $[\![\_]\!] : \mathit{SimpleExp} \to \mathbb{N}$ by induction on the structure of expressions:

$$\begin{aligned} [\![n]\!] &= n \\ [\![E_1 + E_2]\!] &= [\![E_1]\!] \underline{+} [\![E_2]\!] \end{aligned}$$

The semantics is **compostional** in that the meaning of the compound expression $E_1 + E_2$ is given in terms of the meaning of its subexpressions $E_1$ and $E_2$.

**Remarks**

- The semantic domain is entirely separate from the syntax: for example, the set of natural numbers is a mathematical entity in its own right.

- The meaning of the compound term like $E_1 + E_2$ is given in terms of the meanings of its subterms. Hence, we have really given a meaning to the syntactic operation $+$. In this case, the meaning of $+$ is the usual addition function. We call a semantics *compositional* when it has this property, which lets us calculate meanings bit by bit, starting from the numerals and working up. Slide 3 shows an example of a calculation.

**Slide 3**

## Calculating Semantics

$$\begin{aligned}
[\![(1 + (2 + 3))]\!] &= [\![1]\!] \underline{+} [\![(2 + 3)]\!] \\
&= 1 \underline{+} [\![(2 + 3)]\!] \\
&= 1 \underline{+} ([\![2]\!] \underline{+} [\![3]\!]) \\
&= 1 \underline{+} (2 \underline{+} 3) \\
&= 6.
\end{aligned}$$

The denotational semantics for expressions is particularly easy to work with, and much less cumbersome than the operational semantics. For example, it is easy to prove simple facts such as the associativity of the syntactic addition given on slide 4.

Slide 4

## Associativity of addition

**Theorem** For all $E_1$, $E_2$ and $E_3$,

$$[\![E_1 + (E_2 + E_3)]\!] = [\![(E_1 + E_2) + E_3]\!]$$

**Proof**

$$\begin{aligned}
[\![E_1 + (E_2 + E_3)]\!] &= [\![E_1]\!] \pm [\![E_2 + E_3]\!] \\
&= [\![E_1]\!] \pm ([\![E]\!]_2 \pm [\![E_3]\!]) \\
&= ([\![E_1]\!] \pm [\![E]\!]_2) \pm [\![E_3]\!] \\
&= [\![E_1 + E_2]\!] \pm [\![E_3]\!] \\
&= [\![(E_1 + E_2) + E_3)]\!]
\end{aligned}$$

**Exercise** State and prove a similar fact using the big-step semantics.

### Contextual Equivalence

We now introduce an important idea in semantics: that of *contextual equivalence* between programs. Intuitively, we should be able to use equivalent programs (programs that behave in the same way) interchangeably: that is, if $P_1 \cong P_2$ (the symbol $\cong$ means that the programs are equivalent) and $P_1$ is used in some *context* (we write $C[P]$ for program $P$ in program context $C[-]$), then we should get the same effect if we replace $P_1$ with $P_2$: that is, we expect $C[P_1] \cong C[P_2]$. To make this more precise, we say that a *context* $C[-]$ is a program with a *hole* where you would ordinarily expect to see a subprogram.

Slide 5

## Expression Contexts

The set of **expression contexts**, *ContextExp*, is defined by

$$C \in \textit{ContextExp} ::= - \mid E + C \mid C + E \mid \ldots$$

where $E \in$ *ContextExp*.

We say that the symbol $-$ denotes the context hole of the context expression, and write $C[-]$ to emphasise the hole of $C$.

**Exercise** Give a different definition of expression contexts that have zero, one or many holes.

Slide 6

## Some Simple Contexts

$$\begin{aligned} C_1[-] &= - \\ C_2[-] &= - + 2 \\ C_3[-] &= (- + 1) + 3 \\ C_4[-] &= (3 + 4) + - \end{aligned}$$

Slide 7

## Filling the Hole

Given an expression $E$ and context $C[-]$, we can fill the hole with $E$ yielding a new expression, written $C[E]$.

**Context application**, $C[E]$, for expression context $C$ and simple expression $E$, is defintied inductively on the structure of $C$ by:

$$\begin{aligned}(-)[E] &= E\\ (E' + C)[E] &= E' + C[E]\\ (C + E')[E] &= C[E] + E'\end{aligned}$$

Slide 8

## Application Examples

$$\begin{aligned}C_1[3+4] &= 3+4\\ C_2[3+4] &= (3+4)+2\\ C_3[3+4] &= ((3+4)+1)+3\\ C_4[3+4] &= (3+4)+(3+4)\end{aligned}$$

**Slide 9**

## Contextual Equivalence for Expressions

Expressions $E_1$ and $E_2$ are **contextually equivalent** with respect to the big-step semantics if and only if, **for all** contexts $C[-]$ and all numerals $n$,

$$C[E_1] \Downarrow n \Leftrightarrow C[E_2] \Downarrow n.$$

Contextual equivalence for expressions is quite simple. Contextual expressions for programs is more interesting.

For a simple language like *SimpleExp*, contextual equivalence does not mean very much; it turns out that two expressions are contextually equivalent if and only if they have the same final answer. In general though, it is a very important notion. To see this, think of the following two pieces of code which compute factorials.

Slide 10

## Two Factorial Programs

```
int fact(int x){
  int i = 1;
  int j = x;
  while (j > 0){
  i = i * j;
  j = j - 1;
  }
  return i;
}
```

```
int fact(int x){
  if (x <= 0)
   { return 1; }
  else
   {return (x*fact(x - 1));}
 }
```

In ML (with the syntax suitably altered), these programs are contextually equivalent. In Java, they are not.

These two pieces of code do the same thing, in that they each take an integer and return its factorial. Whether these pieces of code are *contextually equivalent* or not, depends on what contexts are available, which depends on the programming language under consideration:in ML (with the syntax suitably altered), they are equivalent. In Java, they are not (it's tricky, think about overriding the `fact( )` method).

### Compositionality and Context Equivalence

Recall that the denotational semantics is *compositional*: that is, the meaning of a large expression is built out of the meanings of its subphrases. It follows that each context determines a 'function between meanings': that is, for each $C[-]$, there is a function $f : \mathbb{N} \to \mathbb{N}$ such that

$$[\![C[E]]\!] = f([\![E]]\!])$$

for any expression $E$. For us, the most important consequence of this is that

$$\text{if } [\![E_1]]\!] = [\![E_2]]\!] \text{ then } [\![C[E_1]]]\!] = [\![C[E_2]]]\!] \text{ for all } C[-].$$

Therefore, if we show

$$[\![E]]\!] = n \text{ if and only if } E \Downarrow n$$

we can use our semantics to reason about contextual equivalence: that is, we will know that denotationally equivalent phrases are in fact contextually equivalent. For *SimpleExp*, this is indeed the case.

**Slide 11**

### Compositionality and Contextual Equivalence

**Theorem**

For arbitrary expression $E$, $[\![E]\!] = n$ if and only if $E \Downarrow n$.

By compositionality of $[\![\_]\!]$, expressions $E_1$ and $E_2$ are contextually equivalent if and only if $[\![E_1]\!] = [\![E_2]\!]$.

This result holds because the denotational semantics is **compositional**.

For more interesting languages, the relationship between the operational and denotational semantics can be more subtle, but the principle of compositionality allows the denotational semantics to be used to reason about contextual equivalence in just the same way.

## Denotational Semantics of *While*

We shall now begin to explore a denotational semantics of our simple language *While*. The first step is to choose our semantic domains.

Slide 12

## Semantic Functions for *While*

We will define the **denotational semantics** of commands, expressions and booleans using the functions of the form

$$\mathcal{C}[\![-]\!] : \texttt{Com} \to ?$$
$$\mathcal{E}[\![-]\!] : \texttt{Exp} \to ?$$
$$\mathcal{B}[\![-]\!] : \texttt{Bool} \to ?$$

We need to choose our semantic domains.

### Semantic Domain for Commands

Let us first focus on commands, which provide the biggest difference between *While* and *SimpleExp*. Let $\Sigma$ be the set of all states. Define $\Sigma_\perp$ to be the set $\Sigma \cup \{\perp\}$: that is, the set $\Sigma$ together with the extra element $\perp$, called *undefined* or *bottom*, which represents the stuck computation or an infinite loop. Then, the semantic domain for commands, called the set of *state transformers*, is defined on slide 13.

Slide 13

## Semantic Domain for *While*

The **semantic domain of commands** is given by the set of **state transformers** defined by

$$\mathtt{ST} = [\Sigma \to \Sigma_\perp] :$$

that is, the set of (total) functions which take a starting state and return either a final state, or the element $\perp$ (called **undefined** or **bottom**), to indicate that the computation got stuck or looped for ever.

**A Note on Notation** The metavariable $s$, and variants of it like $s'$ and so on, will be used to range over *proper states*, not including $\perp$. So, if we say that $f(s) = s'$, it is implicit that $s' \neq \perp$.

### Other Semantic Domains

For expressions and booleans, note that out language allows an expression or a boolean to *depend upon* the store, but *not* to change it. Also, though expressions and booleans cannot get into infinite loops, they may become stuck, so we have to account for this in our choice of semantic domain.

Slide 14

## Semantic Domains for Expressions and Booleans

The *semantic domain of expressions* is

$$\mathtt{E} = [\Sigma \to \mathbb{N}_\perp]$$

The *semantic domain of booleans* is

$$\mathtt{P} = [\Sigma \to \mathbb{B}_\perp]$$

where $\mathbb{B} = \{\mathtt{true}, \mathtt{false}\}$.

**Remark** As before, fixing the semantic domains tells us something about the language. For example, using $\mathtt{ST}$ for the commands acknowledges the possibility of non-termination, but makes clear that the command will yield at most one final state in any given starting state. Similarly, our choice of domain for the booleans automatically eliminates any possibility of side-effects being caused by boolean expressions.

### Semantic Functions

We shall now explore three semantic functions, one for each category in the grammar for *While*. See slide 15.

Slide 15

## Semantic Functions for *While*

$$\mathcal{C}[\![-]\!] : \texttt{Com} \to \texttt{ST}$$

$$\mathcal{E}[\![-]\!] : \texttt{Exp} \to \texttt{E}$$

$$\mathcal{B}[\![-]\!] : \texttt{Bool} \to \texttt{P}$$

**Denotational Semantics of Expressions** The denotational semantics of expressions is defined by induction on the structure of expressions. We shall only give the variable case; the others follow as you would expect from adapting the denotational *SimpleExp*. The case for a variable is simple. The store is examined to see if there is a value for the variable: if so, this value is returned; if not, the expression is stuck and we return $\perp$.

**Slide 16**

## Denotational Semantics of Expressions

The function $\mathcal{E}[\![-]\!] : \mathtt{Exp} \to \mathtt{E}$ is defined inductively by induction on the structure of $\mathtt{Exp}$, with the variable case given by:

$$\begin{aligned} \mathcal{E}[\![x]\!](s) &= s(x) \quad \text{if } s(x) \text{ is defined} \\ &= \perp \quad \text{otherwise} \end{aligned}$$

**Exercise** Do the other cases.

**Denotational Semantics of Booleans** The function $\mathcal{B}[\![-]\!] : \mathtt{Bool} \to \mathtt{P}$ is defined inductively by induction on the structure of $\mathtt{Bool}$. This is straightforward and left as an exercise.

**Denotational Semantics of Commands** The function $\mathcal{C}[\![-]\!] : \mathtt{Com} \to \mathtt{ST}$ is defined inductively by induction on the structure of $\mathtt{Cmd}$. In each case, we ask ourselves how the command transforms the state, and attempt to write down a function which captures our intuition. The next few slides give the definitions. I will not be able to give a full definition (it is difficult), but I will be able to give you the intuition.

Slide 17

## Denotational Semantics of Commands

The function $\mathcal{C}[\![-]\!] : \mathtt{Com} \to \mathtt{ST}$ is defined inductively by induction on the structure of $\mathtt{Com}$, over the next few slides....

Slide 18

## Assignment

The state transformer $\mathcal{C}[\![x := E]\!]$ is defined by

$$\begin{aligned} \mathcal{C}[\![x := E]\!](s) &= s[x \mapsto \mathcal{E}[\![E]\!](s)] \quad \text{if } \mathcal{E}[\![E]\!](s) \neq \bot \\ &= \bot \quad \text{otherwise} \end{aligned}$$

An assignment $x := E$ transforms store $s$ by updating $x$ to contain the value of $E$: Note that, in this definition, $E$ is evaluated in store $s$.

Slide 19

## skip

The state transformer $\mathcal{C}[\![\texttt{skip}]\!]$ is just the **identity** function, defined by

$$\mathcal{C}[\![\texttt{skip}]\!](s) = s$$

`skip` is the easiest command of all. It simply leaves the store alone.

Now consider the case for sequential composition. How does $C_1; C_2$ transform a store? Intuitively, first $C_1$ transforms the original state $s$ to some $s'$, then $C_2$ starts running in state $s'$, leaving some $s''$, which is the outcome of the whole command. If $C_1$ gets stuck or into an infinite loop, so does the whole command; similarly for $C_2$.

Slide 20

## Sequential Composition

The state transformer $\mathcal{C}[\![C_1; C_2]\!]$ is defined by

$$\begin{aligned}\mathcal{C}[\![C_1; C_2]\!](s) &= \bot \quad \text{if } \mathcal{C}[\![C_1]\!](s) = \bot \\ &= \mathcal{C}[\![C_2]\!](\mathcal{C}[\![C_1]\!](s)) \quad \text{otherwise}\end{aligned}$$

Notice that this second line is 'well-typed', because if $\mathcal{C}[\![C_1]\!](s) \neq \bot$ then $\mathcal{C}[\![C_1]\!](s) \in \Sigma$, so we can indeed apply $\mathcal{C}[\![C_2]\!]$ to it.

Slide 21

## Example

We calculate the meaning of the command $C = x := 0; x := x + 1$. For arbitrary state $s$:

$$\begin{aligned}\mathcal{C}[\![C]\!](s) &= (\mathcal{C}[\![x := 0]\!]; \mathcal{C}[\![x := x + 1]\!])(s) \\ &= \mathcal{C}[\![x := x + 1]\!](\mathcal{C}[\![x := 0]\!](s)) \\ &= \mathcal{C}[\![x := x + 1]\!](s[x \mapsto 0]) \\ &= s[x \mapsto \mathcal{E}[\![x + 1]\!](s[x \mapsto 0])].\end{aligned}$$

Since $\mathcal{E}[\![x + 1]\!](s[x \mapsto 0]) = 1$, we have $\mathcal{C}[\![C]\!](s) = s[x \mapsto 1]$.

Slide 22

## Conditional

A command $(\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2)$ transforms a state $s$ as follows:

- work out if $B$ is true or false in state $s$
- if true, transform the state $s$ by running $C_1$
- if false, transform the state $s$ by running $C_2$

Slide 23

## Conditional continued

The state transformer $\mathcal{C}[\![\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]$ is defined by

$$\begin{array}{ll} \mathcal{C}[\![C_1]\!](s) & \text{if } \mathcal{B}[\![B]\!](s) = \texttt{true} \\ \mathcal{C}[\![C_2]\!](s) & \text{if } \mathcal{B}[\![B]\!](s) = \texttt{false} \\ \bot & \text{otherwise} \end{array}$$

**Compositionality**

Recall that we want the denotational semantics to be *compositional*, with the meaning of a program built up out of the meanings of its subprograms. This means that each of the command-forming operations in the language *While* has a denotational meaning. For example, the operation ';', which takes two commands and gives back their sequential composition, has as its meaning the function $\mathtt{seq}$ defined on slide 24. It is reasonable to say that $[\![;]\!] = \mathtt{seq}$. This is really no more than rewriting the original definitions, but it makes the point that denotational semantics gives meaning to the command-forming operations, not just the commands.

**Slide 24**

## The Sequential Composition Operator

We define the function

$$\mathtt{seq} : \mathtt{ST} \times \mathtt{ST} \to \mathtt{ST}$$

by

$$\begin{aligned} \mathtt{seq}(f,g)(s) &= \bot \quad \text{if } f(s) = \bot \\ &= g(f(s)) \quad \text{otherwise} \end{aligned}$$

Slide 25

## The Semantics of the Conditional Operator

We define the function

$$\texttt{cond} : \texttt{P} \times \texttt{ST} \times \texttt{ST} \to \texttt{ST}$$

by

$$\begin{aligned} \texttt{cond}(p, f, g)(s) &= f(s) \quad \text{if } p(s) = \texttt{true} \\ &= g(s) \quad \text{if } p(s) = \texttt{false} \\ &= \bot \quad \text{otherwise} \end{aligned}$$

### Semantics of `while`

Now what about while? How can we write a 'looping' state transformer? Recall the trick that we used to give a small-step semantics to `while`:

$$\langle \texttt{while}\ B\ \texttt{do}\ C, s\rangle \to \langle \texttt{if}\ B\ \texttt{then}\ (C; \texttt{while}\ B\ \texttt{do}\ C)\ \texttt{else skip}, s\rangle.$$

This says that the way $(\texttt{while}\ B\ \texttt{do}\ C)$ transforms the state is the same as the transformation given by

$$\texttt{if}\ B\ \texttt{then}\ (C; \texttt{while}\ B\ \texttt{do}\ C)\ \texttt{else skip}$$

In denotational terms, the statement looks like the equation on slide 26.

Slide 26

## An Equation for `while`

$$\mathcal{C}[\![\texttt{while}\ B\ \texttt{do}\ C]\!] = \mathcal{C}[\![\texttt{if}\ B\ \texttt{then}\ (C;\texttt{while}\ B\ \texttt{do}\ C)\ \texttt{else}\ \texttt{skip}]\!].$$

But can we use this equation as a definition? We are trying to define the semantics of program phrases by induction on their structure. That means, as usual, that when we define the semantics of a compound phrase, we may assume that the semantics of each of its subphrases have already been defined. Bearing this in mind, it is clear that each of the definitions we have given so far is well-defined: that is, the formula on the right-hand side denotes an element of the semantic domain.

The equation above is different. It contains, on the right, a reference to

$$\mathcal{C}[\![\texttt{while}\ B\ \texttt{do}\ C]\!]$$

which we have *not* yet defined: it is not a subphrase of itself! So we have a circular definition, or to put it another way, we do not have a definition at all. However, it does help to think of a while statement as a conditional and analyse how $(\texttt{while}\ B\ \texttt{do}\ C)$ transforms a state $s$.

Slide 27

## Introducing Fixed Points

We need to find some $f \in \mathtt{ST}$ such that

$$f = \mathtt{cond}(\mathcal{B}[\![B]\!], (\mathcal{C}[\![C]\!]; f), \mathtt{id})$$

where `id`, the **identity function**, is the semantic function for `skip` we have already defined. We can then use $f$ as the semantics of `while` $B$ `do` $C$.

Slide 28

## A Helper Function

To put it another way, define a function $F : \mathtt{ST} \to \mathtt{ST}$ by

$$F(f) = \mathtt{cond}(\mathcal{B}[\![B]\!], (\mathcal{C}[\![C]\!]; f), \mathtt{id}).$$

Slide 29

## A First Approximation of `while`

If $B$ is `false` in state $s$, it does nothing: that is, it returns the state $s$.

In this particular case, the transformation is the same as that given by

`if` $B$ `then` **anything** `else skip`

Slide 30

## A Sneaky Step

Since the 'anything' above could be anything (!!), let us replace it with the phrase $(C; \text{anything})$. This gives us

`if` $B$ `then` $(\mathbf{C}; \text{anything})$ `else skip`.

This state transformer is $F(\text{anything})$.

Slide 31

## A Second Approximant

If $B$ is true in state $s$ but becomes false after running the loop body $C$ once, then the loop transforms the state in the same way as

$$\texttt{if}\, B\ \texttt{then}\, C; (\texttt{if}\ B\, \texttt{then}\ C; \textbf{anything}\, \texttt{else skip})\ \texttt{else skip}$$

This state transformer is $F(F(\text{anything}))$.

- If $B$ is false in state $s$, it does nothing: that is, it returns the state $s$. In this particular case, the transformation is the same as that given by

  $$\texttt{if}\, B\, \texttt{then}\, \text{anything}\, \texttt{else skip}.$$

  Since the 'anything' above could be anything, let us replace it with the phrase $(C; \text{anything})$. That gives us

  $$F(\text{anything}) = \texttt{if}\ B\, \texttt{then}\ (C; \text{anything})\, \texttt{else skip}.$$

  where $F$ is the function we defined earlier. This command acts the same way as $(\texttt{while}\ B\, \texttt{do}\ C)$ on those states which do not require entering the loop body at all. Of course, if the loop body is entered, it is very different.

  The key point is that this is a command for which we already have a denotational semantics, and it gives us the right answer some of the time! We shall now improve on this, by finding a command which gives us the right answer more often.

- If $B$ is $\texttt{true}$ in state $s$, $(\texttt{while}\ B\, \texttt{do}\ C)$ runs the command $C$, transforming the state to $s'$; if $B$ is now $\texttt{false}$ then this is the end

of the computation. In this case, therefore, the transformation is the same as that given by

$\texttt{if } B \texttt{ then } (C; \texttt{if } B \texttt{ then anything else skip}) \texttt{ else skip}$

Again replacing anything with $(C; \text{anything})$ gives $F(F(f)) =$

$\texttt{if } B \texttt{ then } (C; \texttt{if } B \texttt{ then } (C; \text{anything}) \texttt{ else skip}) \texttt{ else skip}$

This command gives the right state transformation in the case that the loop body is entered no times, or one time; but if the loop body needs to be entered more than once, it might not be correct. Still, we're getting closer.... Our new command works for all the states the previous one worked for, and some more.

- The command corresponding to $F(F(F(f)))$ gives the correct transformation for states which require going round the loop no times, once or twice. If we write $F^n(f)$ for the command with $n$ uses of $F$, we get the loop $n-1$ or fewer times.

**Slide 32**

## A Sequence of Approximants

Give a starting state $s$:

- if, starting in state $s$, the loop body is executed less than $n$ times, then for any state transformer $f$, $F^n(f)(s)$ gives the same final state that the loop would give;
- if more than $n$ executions of the loop body is required, $F^n(f)(s)$ is right on more and more starting states.

Slide 33

## Better Approximants

Let $f$ to be the state transformer which gives $\perp$ for any starting state $s$. Write this state transformer as $\perp$ too! Then

- if, starting in state $s$, the loop body is executed less than $n$ times, then $F^n(\perp)(s)$ gives the same final state that the loop would give;
- for any natural number $n$ and state $s$, if $F^n(\perp)(s) \neq \perp$ then $F^n(\perp)(s) = F^{n+1}(\perp)(s)$.

**Slide 34**

## We've Got a Fixed Point

Define a state transformer $f$ as follows:

$$\begin{aligned} f(s) &= F^n(\bot)(s) \quad \text{if } F^n(\bot)(s) \neq \bot \text{ for some } n \\ &= \bot \quad \text{otherwise} \end{aligned}$$

This is well-defined and is a fixed point of $F$.

These ideas are enough to let us define a state transformer which gives the fixed point we require. Rather than going into the proof that the $f$ we just defined (on slide 34) is really a state transformer and is really a fixed point of $F$, let us try to do the same tricks using the syntax of *While*.
Consider a program `diverge` which immediately goes into an infinite loop, without changing the state. Add `diverge` as a primitive to our language, just for now, and define

$$\mathfrak{C}[\![\texttt{diverge}]\!](s) = \bot$$

for all states $s$. Then we can define a sequence of *syntactic approximants* to the loop `while` $B$ `do` $C$ as shown in slide 35.

Slide 35

## Approximating a While Loop

We define the approximants of $\texttt{while}\ B\ \texttt{do}\ C$ as follows:

$$
\begin{aligned}
C_0 &= \texttt{diverge} \\
C_1 &= \texttt{if}\ B\ \texttt{then}\ (C; \texttt{diverge})\ \texttt{else skip} \\
&\vdots \\
C_{n+1} &= \texttt{if}\ B\ \texttt{then}\ (C; C_n)\ \texttt{else skip}
\end{aligned}
$$

where `diverge` is a new command which immediately goes into an infinite loop, without changing the state.

(This definition is given by induction on the subscript $i$ of $C$).

We have argued before that the command $C_n$ has the same effect as $(\texttt{while}\ \ B\ \texttt{do}\ \ C)$ in those states which require going fewer than $n$ times round the loop to terminaton. Let us now prove that this sequence of approximations really does get better as $n$ increases.

Slide 36

**Theorem**

For any natural number $n$ and any state $s$, if $\mathcal{C}[\![C_n]\!](s) \neq \perp$ then $\mathcal{C}[\![C_{n+1}]\!](s) = \mathcal{C}[\![C_n]\!](s)$.

**Theorem** For any natural number $n$ and any state $s$, if $\mathcal{C}[\![C_n]\!](s) \neq \perp$ then $\mathcal{C}[\![C_{n+1}]\!](s) = \mathcal{C}[\![C_n]\!](s)$.

**Proof** By induction on $n$.

**Base case:** In the case $n = 0$, $C_n = \texttt{diverge}$ so it is never the case that $\mathcal{C}[\![C_n]\!](s) \neq \perp$. There is therefore nothing to prove.
**Inductive step:** Consider the case $n = k + 1$. By definition,

$$\begin{aligned} \mathcal{C}[\![C_{k+1}]\!] &= \mathcal{C}[\![\texttt{if } B \texttt{ then } (C; C_k) \texttt{ else skip}]\!] \\ &= \texttt{cond}(\mathcal{B}[\![B]\!], (\mathcal{C}[\![C]\!]; \mathcal{C}[\![C_k]\!]), \texttt{id}). \end{aligned}$$

Since we are assuming that $\mathcal{C}[\![C_{k+1}]\!](s) \neq \perp$, it cannot be that $\mathcal{B}[\![B]\!](s) = \perp$. There are therefore two subcases to consider.

- If $\mathcal{B}[\![B]\!](s) = \texttt{false}$, then clearly $\mathcal{C}[\![C_{k+1}]\!](s) = s$, and similarly $\mathcal{C}[\![C_{k+2}]\!](s) = s$, which gives the desired conclusion.
- If $\mathcal{B}[\![B]\!](s) = \texttt{true}$ then $\mathcal{C}[\![C_{k+1}]\!](s) = (\mathcal{C}[\![C]\!]; \mathcal{C}[\![C_k]\!])(s)$. Since we know that this is not $\perp$, it must be the case that

$\mathcal{C}[\![C]\!](s) \neq \perp$, so,

$$\mathcal{C}[\![C_{k+1}]\!](s) = \mathcal{C}[\![C_k]\!](\mathcal{C}[\![C]\!](s)).$$

By the inductive hypothesis,

$$\mathcal{C}[\![C_{k+1}]\!](\mathcal{C}[\![C]\!](s)) = \mathcal{C}[\![C_k]\!](\mathcal{C}[\![C]\!](s)).$$

Putting these two together, we get

$$\mathcal{C}[\![C_{k+1}]\!](s) = \mathcal{C}[\![C_{k+1}]\!](\mathcal{C}[\![C]\!](s)) = \mathcal{C}[\![C_{k+2}]\!](s)$$

Hence, we have proved the result.

■

We therefore have an improving sequence of approximations to our `while` loop. We can now define the semantics of $(\texttt{while}\ B\ \texttt{do}\ C)$ as in slide 37.

**Slide 37**

## Semantics of `while`

$$\begin{aligned} \mathcal{C}[\![\texttt{while}\ B\ \texttt{do}\ C]\!](s) &= s', && \text{if any } \mathcal{C}[\![C_k]\!](s) = s' \\ &= \perp, && \text{otherwise} \end{aligned}$$

It should be reasonably obvious that this is the same state transformer we previously claimed was a fixed point of $F$; see slide 34. The preceding theorem tells us that this does indeed define a function.

Let us now make use of our semantics of `while` to prove a simple fact. Here we will provide that the state left after the execution of a loop always

makes the boolean guard $B$ to be false. We prove this by first showing an appropriate fact about the syntactic approximants to the loop.

**Lemma** For any natural number $n$ and any state $s$, if $\mathcal{C}[\![C_n]\!](s) = s'$ then $\mathcal{B}[\![B]\!](s') = \texttt{false}$.

**Proof** By induction on $n$.

**Base case:** In the case $n = 0$, $C_0 = \texttt{diverge}$ so it never holds that $\mathcal{C}[\![C_n]\!](s) = s'$, so there is nothing to prove.

**Inductive step:** Consider the case $n = k + 1$. By definition,

$$\begin{aligned} \mathcal{C}[\![C_{k+1}]\!] &= \mathcal{C}[\![\texttt{if } B \texttt{ then } (C; C_k) \texttt{ else skip}]\!] \\ &= \texttt{cond}(\mathcal{B}[\![B]\!], (\mathcal{C}[\![C]\!]; \mathcal{C}[\![C_k]\!]), \texttt{id}) \end{aligned}$$

So, if $\mathcal{C}[\![C_{k+1}]\!](s) = s'$, there are two cases:

- either $\mathcal{B}[\![B]\!](x) = \texttt{false}$ and $s = s'$, in which case $\mathcal{B}[\![B]\!](s') = \texttt{false}$ as required, or
- $\mathcal{B}[\![B]\!](s) = \texttt{true}$, and then

  $$s' = (\mathcal{C}[\![C]\!]; \mathcal{C}[\![C_k]\!])(s).$$

  In this case, it is clear that

  $$s' = \mathcal{C}[\![C_k]\!](s'')$$

  where $s'' = \mathcal{C}[\![C]\!](s)$. But the inductive hypothesis tells us that any state coming from $\mathcal{C}[\![C_k]\!]$ makes $\mathcal{B}[\![B]\!]$ false: that is,

  $$\mathcal{B}[\![B]\!](s') = \texttt{false}$$

  as required. ■

**Theorem** If $\mathcal{C}[\![\texttt{while } B \texttt{ do } C]\!](s) = s'$ then $\mathcal{B}[\![B]\!](s') = \texttt{false}$.

**Proof** By definition of the semantics of `while`, if $\mathcal{C}[\![\texttt{while } B \texttt{ do } C]\!](s) = s'$ then $s' = \mathcal{C}[\![C_n]\!](s)$ for some $n$. By the previous lemma, $\mathcal{B}[\![B]\!](s') = \texttt{false}$ as required. ■

**Exercise** Prove that

1. $\mathcal{C}[\![x := y; y := z]\!] = \mathcal{C}[\![x := y]\!]$
2. $\mathcal{C}[\![x := z; y := z]\!] = \mathcal{C}[\![y := z; x := z]\!]$
3. $\mathcal{C}[\![C_1; (C_2; C_3)]\!] = \mathcal{C}[\![(C_1; C_2); C_3]\!]$