Programme

- Introduce a simple language LC for computing on numbers via manipulation of state
Programme

- Introduce a simple language LC for computing on numbers via manipulation of state
- Use this language for introducing the general framework of operational semantics (Just under half the course)
Programme

- Introduce a simple language LC for computing on numbers via manipulation of state
- Use this language for introducing the general framework of operational semantics (Just under half the course)
- Use this experience to understand richer program constructs and languages in remainder of the course
Some notation for LC

- $n \in \mathbb{Z}$ integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
Some notation for LC

- \( n \in \mathbb{Z} \) integers \{\ldots, -2, -1, 0, 1, 2, \ldots\}
- \( b \in B \) booleans \{true, false\}
Some notation for LC

- $n \in Z$ integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
- $b \in B$ booleans $\{\text{true, false}\}$
- $l \in L$ locations or program variables $\{l_0, l_1, l_2, \ldots\}$
Some notation for LC

- $n \in \mathbb{Z}$ integers \{\ldots, −2, −1, 0, 1, 2, \ldots\}
- $b \in B$ booleans \{true, false\}
- $l \in L$ locations or program variables \{\(l_0, l_1, l_2, \ldots\}\)
- $!/l$ - represents integer stored in location \(l\) (dereferencing)
Some notation for LC

- \( n \in \mathbb{Z} \) integers \{\ldots, -2, -1, 0, 1, 2, \ldots\}
- \( b \in B \) booleans \{true, false\}
- \( l \in L \) locations or program variables \{\( l_0, l_1, l_2, \ldots \)\}
- \( l! \) - represents integer stored in location \( l \) (dereferencing)
- \text{iop}: \{+, -, \ast, \ldots\}, operators that take pairs of integers and give an integer result
Some notation for LC

- $n \in \mathbb{Z}$ integers \{\ldots, −2, −1, 0, 1, 2, \ldots\}
- $b \in B$ booleans \{true, false\}
- $l \in L$ locations or program variables \{$l_0, l_1, l_2, \ldots$\}
- $!l$ - represents integer stored in location $l$ (dereferencing)
- iop: \{+, −, *, \ldots\}, operators that take pairs of integers and give an integer result
- bop: \{=, <, >, \ldots\}, operators that take pairs of integers and give a boolean result
LC Syntax

- Phrases
  \[ P ::= C \mid E \mid B \]
LC Syntax

- Phrases
  \[ P ::= C | E | B \]

- Commands
  \[ C ::= \text{skip} | \ell := E | C ; C | \text{if } B \text{ then } C \text{ else } C | \text{while } B \text{ do } C \]
LC Syntax

- Phrases \( P ::= C | E | B \)
- Commands
  \( C ::= \text{skip} | \ell := E | C ; C | \text{if } B \text{ then } C \text{ else } C | \text{while } B \text{ do } C \)
- Integer expressions \( E ::= n | !\ell | E \text{ iop } E \)
Phrases \( P ::= C \mid E \mid B \)

Commands
\( C ::= \text{skip} \mid \ell := E \mid C ; C \mid \text{if } B \text{ then } C \text{ else } C \mid \text{while } B \text{ do } C \)

Integer expressions \( E ::= n \mid \ell \mid E \text{ iop } E \)

Boolean expressions \( B ::= b \mid E \text{ bop } E \)
Example LC Programs

- $z := !x; x := !y; y := !z$  
  $x, y, z$ locations

While

- $l > 1$ do
  - if $l \mod 2 = 0$ then
    - $l := l / 2$
  - else
    - $l := ((3 \times l) + 1) / 2$

Inadequacy of partial correctness?
Example LC Programs

- \( z := !x; x := !y; y := !z \quad \text{x, y, z locations} \)

- while \(!l > 1\) do
  if \(!l \mod 2 = 0\) then \(l := !l/2\) else \(l := ((3*!l) + 1)/2\)
Example LC Programs

- $z := x; x := y; y := z$  \(x, y, z\) locations

- **while** !/ > 1 **do**
  
  **if** !/ mod 2 = 0 **then** !/ := !/ / 2 **else** !/ := (3*!/ + 1) / 2

- Inadequacy of partial correctness?
SMC machines

Historically, first approach to mathematically rigorous operational semantics for programming languages uses abstract SMC machine

- $S$ stack of intermediate and final results
SMC machines

Historically, first approach to mathematically rigorous operational semantics for programming languages uses abstract SMC machine

- **S** stack of intermediate and final results
- **M** memory association of integers with locations
SMC machines

Historically, first approach to mathematically rigorous operational semantics for programming languages uses abstract SMC machine

- **S** stack of intermediate and final results
- **M** memory association of integers with locations
- **C** control stack of phrases to be evaluated
SMC-machine configurations

are triples \( \langle c, r, s \rangle \) consisting of

1. a Control stack \( c \)

\[
c ::= \text{nil} \mid P \cdot c \mid iop \cdot c \mid bop \cdot c \mid := \cdot c \mid \text{if} \cdot c \mid \text{while} \cdot c
\]
SMC-machine configurations

are triples \( \langle c, r, s \rangle \) consisting of

- a Control stack \( c \)

\[
c ::= \text{nil} \mid P \cdot c \mid iop \cdot c \mid bop \cdot c \mid ::= \cdot c \mid \text{if} \cdot c \mid \text{while} \cdot c
\]

- a Stack of (intermediate and final) results

\[
r ::= \text{nil} \mid P \cdot r \mid \ell \cdot r
\]
SMC-machine configurations

are triples \( \langle c, r, s \rangle \) consisting of

- a Control stack \( c \)
  
  \[
c ::= \text{nil} \mid P \cdot c \mid iop \cdot c \mid bop \cdot c \mid := \cdot c \mid \text{if} \cdot c \mid \text{while} \cdot c
  \]

- a Stack of (intermediate and final) results \( r \)
  
  \[
r ::= \text{nil} \mid P \cdot r \mid \ell \cdot r
  \]

- a Memory state, \( s \), which is a partial function mapping locations to integers, defined only at finitely many locations.
Transition system

- Configurations \( <c, r, s> \)
Transition system

- Configurations $< c, r, s >$
- Initial configurations $< C \cdot \text{nil}, \text{nil}, s >$
Transition system

- Configurations \(< c, r, s >\)
- Initial configurations \(< C \cdot \text{nil}, \text{nil}, s >\)
- \(s\) is a memory \(\{l_1 \mapsto n_1, \ldots, l_m \mapsto n_m\}\)

Precise definition of what it means to say "\(C\) executed in state \(s\) terminates successfully producing state \(s'\)"
Transition system

- Configurations $< c, r, s >$
- Initial configurations $< C \cdot \text{nil}, \text{nil}, s >$
- $s$ is a memory $\{l_1 \mapsto n_1, \ldots, l_m \mapsto n_m\}$
- Terminal configurations $< \text{nil}, \text{nil}, s >$
Transition system

- Configurations \(< c, r, s >\)
- Initial configurations \(< C \cdot \text{nil}, \text{nil}, s >\)
- \(s\) is a memory \(\{l_1 \mapsto n_1, \ldots, l_m \mapsto n_m\}\)
- Terminal configurations \(< \text{nil}, \text{nil}, s >\)
- Existence of a run

\[
< C \cdot \text{nil}, \text{nil}, s > \rightarrow^* < \text{nil}, \text{nil}, s' >
\]

precise definition of what it means to say “C executed in state \(s\) terminates successfully producing state \(s'\)”
Transition system

- Configurations $< c, r, s >$
- Initial configurations $< C \cdot \text{nil}, \text{nil}, s >$
- $s$ is a memory $\{l_1 \mapsto n_1, \ldots, l_m \mapsto n_m\}$
- Terminal configurations $< \text{nil}, \text{nil}, s >$
- Existence of a run

$< C \cdot \text{nil}, \text{nil}, s > \xrightarrow{*} < \text{nil}, \text{nil}, s' >$

precise definition of what it means to say “C executed in state $s$ terminates successfully producing state $s'$”

- Semantics given by axioms for transitions
Notation

Assume $s = \{ l_1 \mapsto n_1, \ldots, l_m \mapsto n_m \}$

$\Rightarrow s(l_i) = n_i$ value at $l_i$
Notation

Assume $s = \{l_1 \mapsto n_1, \ldots, l_m \mapsto n_m\}$

- $s(l_i) = n_i$ value at $l_i$
- $s[l \mapsto n]$ update $s$ only at $l$
  - if $l$ is new (different from $l_1, \ldots, l_m$) then
    \[
s[l \mapsto n] = \{l_1 \mapsto n_1, \ldots, l_m \mapsto n_m, l \mapsto n\}
    \]
  - else $l = l_i$ and so
    \[
s[l \mapsto n] = \{l_1 \mapsto n_1, \ldots, l_i \mapsto n, \ldots, l_m \mapsto n_m\}
    \]
Transition rules

Integer expressions

- Constant $\langle n \cdot c, r, s \rangle \rightarrow \langle c, n \cdot r, s \rangle$

Boolean expressions

- Constant $\langle b \cdot c, r, s \rangle \rightarrow \langle c, b \cdot r, s \rangle$
Transition rules

Integer expressions

- **Constant** \(< n \cdot c, r, s > \rightarrow < c, n \cdot r, s >\)
- **Location** \(< !\ell \cdot c, r, s > \rightarrow < c, n \cdot r, s >\) if \(s(\ell) = n\)

Boolean expressions

- **Constant** \(< b \cdot c, r, s > \rightarrow < c, b \cdot r, s >\)
Transition rules

Integer expressions

- **Constant** \(\langle n \cdot c, r, s \rangle \rightarrow \langle c, n \cdot r, s \rangle\)
- **Location** \(\langle !\ell \cdot c, r, s \rangle \rightarrow \langle c, n \cdot r, s \rangle\) if \(s(\ell) = n\)
- **Compound** \(\langle (E_1 \ iop \ E_2) \cdot c, r, s \rangle \rightarrow \langle E_1 \cdot E_2 \cdot iop \cdot c, r, s \rangle\)

Boolean expressions

- **Constant** \(\langle b \cdot c, r, s \rangle \rightarrow \langle c, b \cdot r, s \rangle\)
- **Compound**
  \(\langle (E_1 \ bop \ E_2) \cdot c, r, s \rangle \rightarrow \langle E_1 \cdot E_2 \cdot bop \cdot c, r, s \rangle\)
Transition rules

Integer expressions

- **Constant**  $< n \cdot c, r, s > \rightarrow < c, n \cdot r, s >$

- **Location**  $< !\ell \cdot c, r, s > \rightarrow < c, n \cdot r, s >$ if $s(\ell) = n$

- **Compound**  $< (E_1 \ iop \ E_2) \cdot c, r, s > \rightarrow < E_1 \cdot E_2 \cdot iop \cdot c, r, s >$

- **Operator**  $< iop \cdot c, n_2 \cdot n_1 \cdot r, s > \rightarrow < c, n \cdot r, s >$ if $n_1 \ iop \ n_2 = n$

Boolean expressions

- **Constant**  $< b \cdot c, r, s > \rightarrow < c, b \cdot r, s >$

- **Compound**  $< (E_1 \ bop \ E_2) \cdot c, r, s > \rightarrow < E_1 \cdot E_2 \cdot bop \cdot c, r, s >$

- **Operator**  $< bop \cdot c, n_2 \cdot n_1 \cdot r, s > \rightarrow < c, b \cdot r, s >$ if $n_1 \ bop \ n_2 = b$
Transition rules for commands I

- **Skip** $< \text{skip} \cdot c, r, s > \rightarrow < c, r, s >$
Transition rules for commands I

- **Skip** \(< \text{skip} \cdot c, r, s > \rightarrow < c, r, s >\)
- **Assignment** \(< (\ell := E) \cdot c, r, s > \rightarrow < E := \cdot c, \ell \cdot r, s >\)
Transition rules for commands

- **Skip** \( \langle \text{skip} \cdot c, r, s \rangle \longrightarrow \langle c, r, s \rangle \)
- **Assignment** \( \langle \ell := E \cdot c, r, s \rangle \longrightarrow \langle E := \cdot c, \ell \cdot r, s \rangle \)
- **Assign** \( \langle := \cdot c, n \cdot \ell \cdot r, s \rangle \longrightarrow \langle c, r, s[\ell \mapsto n] \rangle \)
Transition rules for commands I

- **Skip**  
  \[ < \text{skip} \cdot c, r, s > \rightarrow < c, r, s > \]

- **Assignment**  
  \[ < (\ell := E) \cdot c, r, s > \rightarrow < E \cdot := \cdot c, \ell \cdot r, s > \]

- **Assign**  
  \[ < := \cdot c, n \cdot \ell \cdot r, s > \rightarrow < c, r, s[\ell \mapsto n] > \]

- **Sequencing**  
  \[ < (C_1 ; C_2) \cdot c, r, s > \rightarrow < C_1 \cdot C_2 \cdot c, r, s > \]
Transition rules for commands II

- **Conditional**
  $<(\text{if } B \text{ then } C_1 \text{ else } C_2) \cdot c, r, s> \rightarrow <B \cdot \text{if} \cdot c, C_1 \cdot C_2 \cdot r, s>$

- **Iteration**
  $<(\text{while } B \text{ do } C) \cdot c, r, s> \rightarrow <B \cdot \text{while} \cdot c, B \cdot C \cdot r, s>$
Transition rules for commands II

- **Conditional**
  \[
  < (\text{if } B \text{ then } C_1 \text{ else } C_2) \cdot c, r, s > \longrightarrow < B \cdot \text{if } c, C_1 \cdot C_2 \cdot r, s >
  \]

- **If-True**
  \[
  < \text{if } c, \text{true} \cdot C_1 \cdot C_2 \cdot r, s > \longrightarrow < C_1 \cdot c, r, s >
  \]

- **If-False**
  \[
  < \text{if } c, \text{false} \cdot C_1 \cdot C_2 \cdot r, s > \longrightarrow < C_2 \cdot c, r, s >
  \]

- **Iteration**
  \[
  < (\text{while } B \text{ do } C) \cdot c, r, s > \longrightarrow < B \cdot \text{while } c, B \cdot C \cdot r, s >
  \]
Transition rules for commands II

- **Conditional**
  \[
  < (\text{if } B \text{ then } C_1 \text{ else } C_2) \cdot c, r, s > \rightarrow < B \cdot \text{if} \cdot c, C_1 \cdot C_2 \cdot r, s >
  \]

- **If-True**
  \[
  < \text{if} \cdot c, \text{true} \cdot C_1 \cdot C_2 \cdot r, s > \rightarrow < C_1 \cdot c, r, s >
  \]

- **If-False**
  \[
  < \text{if} \cdot c, \text{false} \cdot C_1 \cdot C_2 \cdot r, s > \rightarrow < C_2 \cdot c, r, s >
  \]

- **Iteration**
  \[
  < (\text{while } B \text{ do } C) \cdot c, r, s > \rightarrow < B \cdot \text{while} \cdot c, B \cdot C \cdot r, s >
  \]

- **While-True**
  \[
  < \text{while} \cdot c, \text{true} \cdot B \cdot C \cdot r, s > \rightarrow < C \cdot (\text{while } B \text{ do } C) \cdot c, r, s >
  \]

- **While-FALSE**
  \[
  < \text{while} \cdot c, \text{false} \cdot B \cdot C \cdot r, s > \rightarrow < c, r, s >
  \]
\[ \langle C \cdot \text{nil}, \text{nil}, s \rangle \quad \text{(Iteration)} \]
\[ \rightarrow \langle B \cdot \text{while} \cdot \text{nil}, B \cdot C' \cdot \text{nil}, s \rangle \quad \text{(Compound)} \]
\[ \rightarrow \langle !\ell \cdot 0 \cdot > \cdot \text{while} \cdot \text{nil}, B \cdot C' \cdot \text{nil}, s \rangle \quad \text{(Location)} \]
\[ \rightarrow \langle 0 \cdot > \cdot \text{while} \cdot \text{nil}, 4 \cdot B \cdot C' \cdot \text{nil}, s \rangle \quad \text{(Constant)} \]
\[ \rightarrow \langle > \cdot \text{while} \cdot \text{nil}, 0 \cdot 4 \cdot B \cdot C' \cdot \text{nil}, s \rangle \quad \text{(Operator)} \]
\[ \rightarrow \langle \text{while} \cdot \text{nil}, \text{true} \cdot B \cdot C' \cdot \text{nil}, s \rangle \quad \text{(While-True)} \]
\[ \rightarrow \langle C' \cdot C \cdot \text{nil}, \text{nil}, s \rangle \]
\[ \rightarrow^* \langle \text{nil}, \text{nil}, s[\ell \mapsto 0, \ell' \mapsto 24] \rangle \]

where
\[
\begin{align*}
C & \stackrel{\text{def}}{=} \text{while } B \text{ do } C', \\
B & \stackrel{\text{def}}{=} !\ell > 0, \\
C' & \stackrel{\text{def}}{=} \ell' := !\ell * !\ell'; \ell := !\ell - 1, \\
s & \stackrel{\text{def}}{=} \{\ell \mapsto 4, \ell' \mapsto 1}\).
\end{align*}
\]
But...

- SMC machine is not very intuitive
But ...

- SMC machine is not very intuitive
- Only a few transitions perform computation. Others are just concerned with phrase analysis
But...

- SMC machine is not very intuitive
- Only a few transitions perform computation. Others are just concerned with phrase analysis
- Many “useless” configurations that can never be reached from an initial configuration

\[
< \text{while} \cdot c, 42 \cdot B \cdot C \cdot r, s >
\]
But ...

- SMC machine is not very intuitive
- Only a few transitions perform computation. Others are just concerned with phrase analysis
- Many "useless" configurations that can never be reached from an initial configuration

\[ \text{< while } \cdot c, 42 \cdot B \cdot C \cdot r, s > \]

- Doesn't directly formalise our understanding of LC control constructs such as that for while
But . . .

- SMC machine is not very intuitive
- Only a few transitions perform computation. Others are just concerned with phrase analysis
- Many “useless” configurations that can never be reached from an initial configuration

\[
\langle \text{while} \cdot c, 42 \cdot B \cdot C \cdot r, s \rangle
\]

- Doesn’t directly formalise our understanding of LC control constructs such as that for while
- Hard to use SMC machine as a basis for formal reasoning about properties of LC programs. Lose program structure in the rules
Informal Semantics

Here is the informal definition of

\[ \textbf{while } B \textbf{ do } C \]

Informal Semantics

Here is the informal definition of

\[ \text{while } B \text{ do } C \]


- The command \( C \) is executed repeatedly so long as the value of the expression \( B \) remains \textbf{true}. The test takes place before each execution of the command.
Aims of Structural Operational Semantics

- Transition systems should be structured in a way that reflects the structure of the language: the possible transitions for a compound phrase should be built up inductively from the transitions for its constituent subphrases.
Aims of Structural Operational Semantics

- Transition systems should be structured in a way that reflects the structure of the language: the possible transitions for a compound phrase should be built up inductively from the transitions for its constituent subphrases.

- At the same time one tries to increase the clarity of semantic definitions by minimising the role of ad hoc, phrase-analysis transitions and by making the configurations of the transition system as simple (abstract) as possible.