Welcome

- **Lectures**: Monday and Thursday 12.10-13.00
- Course materials on web page
  - Online lecture notes by Andrew Pitts (Cambridge): covers over half the course
  - Further reading materials
  - Background reading (such as Plotkin’s original Aarhus notes on SOS)
  - Slides (such as this one)
- Exam counts for 80% of course and Coursework for 20%
- Two equally weighted handins for Coursework
- Weekly tutorials (starting week 3)
Constituents of programming language definition

- Syntax: the alphabet of symbols and a formal description of the well-formed expressions, phrases, programs, etc.
Constituents of programming language definition

- **Syntax**: the alphabet of symbols and a formal description of the well-formed expressions, phrases, programs, etc.
- **Pragmatics**: description and examples of how the various features of the language are intended to be used.
  Implementation of the language (compilers and interpreters).
  Auxiliary tools (syntax checkers, debuggers, etc.).
Constituents of programming language definition

- **Syntax:** the alphabet of symbols and a formal description of the well-formed expressions, phrases, programs, etc.
- **Pragmatics:** description and examples of how the various features of the language are intended to be used. Implementation of the language (compilers and interpreters). Auxiliary tools (syntax checkers, debuggers, etc.).
- **Semantics:** The meaning of the language’s features (e.g. their run-time behaviour)—all too often only specified informally, or via the previous heading.
Finally the procedure body, modified as above, is inserted in place of the procedure statement and executed. If a procedure is called from a place outside the scope of any nonlocal quantity of the procedure body, the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers.
Uses of formal, mathematical semantics

- **Implementation issues**: Machine-independent specification of behaviour. Correctness of program analyses and optimisations.

  - Mathematical tools used for semantics can suggest useful new programming styles. (E.g. influence of Church's lambda calculus (circa 1934) on functional programming)
Uses of formal, mathematical semantics

- **Implementation issues**: Machine-independent specification of behaviour. Correctness of program analyses and optimisations.
- **Verification**: Basis of methods for reasoning about program properties and program specifications.

Mathematical tools used for semantics can suggest useful new programming styles. (E.g. influence of Church's lambda calculus (circa 1934) on functional programming)
Uses of formal, mathematical semantics

- Implementation issues: Machine-independent specification of behaviour. Correctness of program analyses and optimisations.
- Verification: Basis of methods for reasoning about program properties and program specifications.
- Language design: Can bring to light ambiguities and unforeseen subtleties in programming language constructs. Mathematical tools used for semantics can suggest useful new programming styles. (E.g. influence of Church’s lambda calculus (circa 1934) on functional programming)
Styles of semantics

- **Denotational**: Meanings for program phrases defined abstractly as elements of some suitable mathematical structure.
Styles of semantics

- **Denotational**: Meanings for program phrases defined abstractly as elements of some suitable mathematical structure.
- **Axiomatic**: Meanings for program phrases defined indirectly via the axioms and rules of some logic of program properties.
Styles of semantics

- **Denotational**: Meanings for program phrases defined abstractly as elements of some suitable mathematical structure.
- **Axiomatic**: Meanings for program phrases defined indirectly via the axioms and rules of some logic of program properties.
- **Operational**: Meanings for program phrases defined in terms of the steps of computation they can take during program execution.
Example: \( z := x; x := y; y := z \)

**Denotational view**

- meaning is “abstract”
Example: $z := x; x := y; y := z$

Denotational view

- meaning is “abstract”
- Notion of a Memory as an association of variables with values.

$$\{x \mapsto 4, y \mapsto 8, \ldots\}$$
Example:  \( z := x; x := y; y := z \)

Denotational view

- meaning is “abstract”
- Notion of a Memory as an association of variables with values.
  
  \[
  \{ x \mapsto 4, y \mapsto 8, \ldots \}
  
  \]
- meaning of fragment is the function

  \[
  f : \text{Memory} \rightarrow \text{Memory}
  
  \text{where } f(M) = M' \text{ when } M' \text{ is the same as } M \text{ except that the values of } x, y, z \text{ in } M' \text{ are those of } y, x, x \text{ in } M.
  \]
Example cont: $z := x; x := y; y := z$

Axiomatic view

- One approach partial correctness with respect to pre and post conditions
Example cont: \( z := x; x := y; y := z \)

Axiomatic view

- One approach partial correctness with respect to pre and post conditions

\[
\{ \text{formula} \} \ p \ \{ \text{formula’} \}
\]

“if formula is true before execution of \( p \), then formula’ is true after execution of \( p \)”
Example cont: $z := x; x := y; y := z$

Axiomatic view

- One approach partial correctness with respect to pre and post conditions
- $\{ \text{formula} \} \ p \ \{ \text{formula'} \}$
  “if formula is true before execution of $p$, then formula’ is true after execution of $p$”
- For instance,

  $\begin{align*}
  \{x = n \land y = m\} & \quad z := x & \quad \{z = n \land y = m\} \\
  \{z = n \land y = m\} & \quad x := y & \quad \{z = n \land x = m\} \\
  \{z = n \land x = m\} & \quad y := z & \quad \{y = n \land x = m\}
  \end{align*}$
Example cont: \( z := x; x := y; y := z \)

Axiomatic view

- One approach partial correctness with respect to pre and post conditions

- \{ \text{formula} \} \ p \ \{ \text{formula’} \} 

  “if \text{formula} \ is \ true \ before \ execution \ of \ p, \ then \ \text{formula’} \ is \ true \ after \ execution \ of \ p”

- For instance,

\[
\begin{align*}
\{x = n \land y = m\} & \quad z := x & \quad \{z = n \land y = m\} \\
\{z = n \land y = m\} & \quad x := y & \quad \{z = n \land x = m\} \\
\{z = n \land x = m\} & \quad y := z & \quad \{y = n \land x = m\}
\end{align*}
\]

- Putting them together

\[
\{x = n \land y = m\} z := x; x := y; y := z \{y = n \land x = m\}
\]
Example cont: $z := x; x := y; y := z$

Operational view

- "abstract" machine view (the approach adopted in this course)
Example cont: \( z := x; x := y; y := z \)

Operational view

- “abstract” machine view (the approach adopted in this course)
- “Configuration” plus abstract “execution relation”

\[
(z := x; x := y; y := z, \{x \mapsto n, y \mapsto m, \ldots\}) \Downarrow
(x := y; y := z, \{x \mapsto n, y \mapsto m, z \mapsto n, \ldots\}) \Downarrow
(y := z, \{x \mapsto m, y \mapsto m, z \mapsto n, \ldots\}) \Downarrow
(\epsilon, \{x \mapsto m, y \mapsto n, z \mapsto n, \ldots\})
\]
Example cont:  $z := x; x := y; y := z$  

Operational view

- “abstract” machine view (the approach adopted in this course)
- “Configuration” plus abstract “execution relation”

$$(z := x; x := y; y := z, \{x \mapsto n, y \mapsto m, \ldots\}) \Downarrow\rightarrow \ldots$$

$$(x := y; y := z, \{x \mapsto n, y \mapsto m, z \mapsto n, \ldots\}) \Downarrow\rightarrow \ldots$$

$$(y := z, \{x \mapsto m, y \mapsto m, z \mapsto n, \ldots\}) \Downarrow\rightarrow \ldots$$

$$(\varepsilon, \{x \mapsto m, y \mapsto n, z \mapsto n, \ldots\})$$

- Main ingredient: transition system
A transition system is specified by

- a set $\text{Config}$ of configurations/states, and
Transition systems defined

A transition system is specified by

• a set $\textit{Config}$ of configurations/states, and
• a binary relation $\rightarrow {\subseteq \textit{Config} \times \textit{Config}}$. 
Transition systems defined

A transition system is specified by

- a set \( \text{Config} \) of configurations/states, and
- a binary relation \( \rightarrow \subseteq \text{Config} \times \text{Config} \).

- the binary relation is written infix, i.e. \( c \rightarrow c' \) means \( c \) and \( c' \) are related by \( \rightarrow \).
Exercise

What would be the denotational/axiomatic/operational meanings of

- $x := x + 1$?
- if $x = 0$ then $y := z$ else $z := y$?
- while $x > 0$ do $y := y + x$; $x := x - 1$?
Notation for transition system \((\text{Config}, \rightarrow)\)

\(\rightarrow^*\) is reflexive-transitive closure of \(\rightarrow\)

Often augment with distinguished subsets \(I\) and \(T\), \((\text{Config}, \rightarrow, I, T)\) where \(I\): initial configurations and \(T\): terminal configurations

Run of a transition system: \(i \rightarrow^* t, i \in I, t \in T\) (Often if \(t \in T\), then not(\(t \rightarrow\)))

if \(c \notin T\) has the property not(\(c \rightarrow\)) then \(c\) is stuck/deadlocked
Notation for transition system \((\text{Config}, \rightarrow)\)

- \(\rightarrow^*\) is reflexive-transitive closure of \(\rightarrow\)
- \(c \rightarrow\) shorthand for \(c \rightarrow c'\) for some \(c'\)

- Often augment with distinguished subsets \(I\) and \(T\), \((\text{Config}, \rightarrow, I, T)\) where \(I\): initial configurations and \(T\): terminal configurations

- Run of a transition system: \(i \rightarrow^* t\), \(i \in I\) \(t \in T\) (Often if \(t \in T\), then not(\(t \rightarrow\)))

- if \(c \notin T\) has the property not(\(c \rightarrow\)) then \(c\) is stuck/deadlocked
Notation for transition system \((\text{Config}, \rightarrow)\)

- \(\rightarrow^*\) is reflexive-transitive closure of \(\rightarrow\)
- \(c \rightarrow\) shorthand for \(c \rightarrow c'\) for some \(c'\)
- \(\rightarrow\) is deterministic: for all configs \(c, c_1, c_2\)

\[
\text{if } c \rightarrow c_1 \text{ and } c \rightarrow c_2, \text{ then } c_1 = c_2
\]
Notation for transition system \((\text{Config, } \rightarrow)\)

- \(\rightarrow^*\) is reflexive-transitive closure of \(\rightarrow\)
- \(c \rightarrow\) shorthand for \(c \rightarrow c'\) for some \(c'\)
- \(\rightarrow\) is deterministic: for all configs \(c, c_1, c_2\)
  \[ \text{if } c \rightarrow c_1 \text{ and } c \rightarrow c_2, \text{ then } c_1 = c_2 \]
- Often augment with distinguished subsets \(I\) and \(T\), \((\text{Config, } \rightarrow, I, T)\) where \(I\): initial configurations and \(T\): terminal configurations
Notation for transition system \((\text{Config}, \rightarrow)\)

- \(\rightarrow^*\) is reflexive-transitive closure of \(\rightarrow\)
- \(c \rightarrow\) shorthand for \(c \rightarrow c'\) for some \(c'\)
- \(\rightarrow\) is deterministic: for all configs \(c, c_1, c_2\)
  
  \[
  \text{if } c \rightarrow c_1 \text{ and } c \rightarrow c_2, \text{ then } c_1 = c_2
  \]
- Often augment with distinguished subsets \(I\) and \(T\), \((\text{Config}, \rightarrow, I, T)\) where \(I\): initial configurations and \(T\): terminal configurations
- Run of a transition system: \(i \rightarrow^* t, i \in I \ t \in T\) (Often if \(t \in T\), then not\(t \rightarrow\))
Notation for transition system (\(\text{Config}, \rightarrow\))

- \(\rightarrow^*\) is reflexive-transitive closure of \(\rightarrow\)
- \(c \rightarrow\) shorthand for \(c \rightarrow c'\) for some \(c'\)
- \(\rightarrow\) is deterministic: for all configs \(c, c_1, c_2\)
  
  \[
  \text{if } c \rightarrow c_1 \text{ and } c \rightarrow c_2, \text{ then } c_1 = c_2
  \]

- Often augment with distinguished subsets \(I\) and \(T\), (\(\text{Config}, \rightarrow, I, T\)) where \(I\): initial configurations and \(T\): terminal configurations
- Run of a transition system: \(i \rightarrow^* t, i \in I \ t \in T\) (Often if \(t \in T\), then not \((t \rightarrow))\)
- if \(c \notin T\) has the property not \((c \rightarrow)\) then \(c\) is stuck/deadlocked