Equivalence

- Two expressions (commands) are equivalent if they yield the same result from all states:
  \[ e_1 = e_2 \iff \forall \sigma \in \Sigma. \forall n \in \mathbb{Z}. <e_1, \sigma> \Downarrow n \iff <e_2, \sigma> \Downarrow n \]

and for commands:

- Two commands are equivalent if they yield the same result from all states:
  \[ c_1 = c_2 \iff \forall \sigma, \sigma' \in \Sigma. <c_1, \sigma> \Downarrow \sigma' \iff <c_2, \sigma> \Downarrow \sigma' \]

Proving an Equivalence

- Prove that "skip; c ≈ c" for all c.
- Assume that \( \text{D} :: <\text{skip}; c, \sigma> \Downarrow \sigma' \).
- By inversion (twice) we have that:
  \[ \text{D} :: <\text{c}, \sigma> \Downarrow \sigma' \]
  \[ \text{D}_1 :: <\text{c}, \sigma> \Downarrow \sigma' \]
- Thus, we have \( \text{D}_1 :: <\text{c}, \sigma> \Downarrow \sigma' \).
- The other direction is similar.

Proving an Inequivalence

- Prove that \( x := y \not\equiv x := z \) when \( y \neq z \).
- It suffices to produce a witness store \( \sigma \) in which the two commands yield different results:
  \[ \sigma(y) = 0 \text{ and } \sigma(z) = 1. \]
  \[ \text{Then } <x := y, \sigma> \Downarrow \sigma[x := 0] \]
  \[ \text{and } <x := z, \sigma> \Downarrow \sigma[x := 1]. \]

Operational Semantics: Big-Step vs. Small-Step

- Big-Step Operational Semantics: \( e \Downarrow n \)
  - Judgment \( e \Downarrow n \) means that \( e \) evaluates to \( n \).
  - In one big step, all the way to a result.
  - Hard to talk about commands that do not terminate.
    - There is no \( n' \) such that \( c \Downarrow n' \).
    - But we do not have an explanation of how \( c \) runs or fails.
  - It does not give us a way to talk about intermediate states.
    - Thus we cannot say that on a parallel machine the execution of two commands is interleaved.
- Small-Step Operational Semantics: \( e \rightarrow e' \)
  - describes a single step in the evaluation.
  - Many steps may be needed to get a result.

What is the relation \( \rightarrow \) defined by these rules?

- \( n \) is the sum of \( n_1 \) and \( n_2 \)
  \[ n_1 + n_2 \rightarrow n \]
  \[ e_1 \rightarrow e_1' \]
  \[ e_2 \rightarrow e_2' \]
  \[ e_1 + e_2 \rightarrow e_1' + e_2' \]
  \[ e_1 \rightarrow e_1' \]
  \[ e_2 \rightarrow e_2' \]
  \[ e_1 + e_2 \rightarrow e_1' + e_2' \]

- \( n \) is the product of \( n_1 \) and \( n_2 \)
  \[ n_1 \cdot n_2 \rightarrow n \]
  \[ e_1 \rightarrow e_1' \]
  \[ e_2 \rightarrow e_2' \]
  \[ n_1 \cdot e_2 \rightarrow n_1 \cdot e_2' \]
  \[ e_1 \rightarrow e_1' \]
  \[ e_2 \rightarrow e_2' \]
  \[ n_1 \cdot e_2 \rightarrow n_1 \cdot e_2' \]
Small-Step Evaluation Rules

- $n$ is the sum of $n_1$ and $n_2$, $n_1 + n_2 \rightarrow n$
- $n_1 \cdot n_2 \rightarrow n$
- $e_1 + e_2 \rightarrow e_1' + e_2$
- $e_1 \cdot e_2 \rightarrow e_1' \cdot e_2$
- $e_1 + e_2 \rightarrow e_1' + e_2$
- $e_1 \cdot e_2 \rightarrow e_1' \cdot e_2$

- Fixed evaluation order.
- Example: $(3 + 4) + 5 \rightarrow 7 + 5 \rightarrow 12$

Contextual Semantics

- Contextual semantics is a small-step semantics that is specified in two parts:
  - What evaluation rules to apply?
    - What is an atomic reduction step?
  - Where can we apply them?
    - Where should we apply the next atomic reduction step?

Small-Step Operational Semantics for IMP

- Each execution step is a rewrite of the program.
- We will define a relation $\langle c, \sigma \rangle \rightarrow \langle e, \sigma' \rangle$
  - $c'$ is obtained from $c$ through an atomic rewrite step.
  - E.g.: $x := 2+8, \sigma \rightarrow \langle x := 10, \sigma \rangle \rightarrow \langle \text{skip}, \sigma[x:=10] \rangle$
  - Evaluation terminates when the program has been rewritten to a terminal program (one from which we cannot make further progress).
  - For IMP the terminal command is "skip".
  - As long as the command is not "skip" we can make progress.
  - Some commands never reduce to skip (e.g., while true do skip).

What is an Atomic Reduction?

- We need to define:
  - What constitutes an atomic reduction step?
    - Granularity is a choice of the semantics designer.
      - E.g., choice between an addition of arbitrary integers, or an addition of 32-bit integers.
    - How to select the next reduction step, when several are possible?
      - This is the order of evaluation issue.

Redexes

- A redex is a syntactic expression or command that can be reduced (transformed) in one atomic step.
- For brevity, we mix expression and command redexes (and also omit some redexes and contexts).
- Redexes are defined by a grammar:
  
  $$ r ::= x | n_1 \cdot n_2 | x := n | \text{skip} c | \text{if true then } c_1 \text{ else } c_2 | \text{if false then } c_1 \text{ else } c_2 $$

- Note that $(1 + 3) \cdot 2$ is not a redex, but $1 + 3$ is.

Local Reduction Rules for IMP

- One for each redex $\langle r, \sigma \rangle \rightarrow \langle e, \sigma' \rangle$
  - This means that in state $\sigma$, the redex $r$ can be replaced in one step with the expression $e$.  
    - $\langle x, \sigma \rangle \rightarrow \langle \sigma(x), \sigma \rangle$
    - $\langle n_1 + n_2, \sigma \rangle \rightarrow \langle n, \sigma \rangle$ where $n = n_1 + n_2$
    - $\langle n_1 \cdot n_2, \sigma \rangle \rightarrow \langle \text{true}, \sigma \rangle$ if $n_1 \cdot n_2$
    - $\langle x := n_1, \sigma \rangle \rightarrow \langle \text{skip}, \sigma[x := n_1] \rangle$
    - $\langle \text{skip}, c, \sigma \rangle \rightarrow \langle c, \sigma \rangle$
    - $\langle \text{if true then } c_1 \text{ else } c_2, \sigma \rangle \rightarrow \langle c_1, \sigma \rangle$
    - $\langle \text{if false then } c_1 \text{ else } c_2, \sigma \rangle \rightarrow \langle c_2, \sigma \rangle$
    - While $b$ do $c$, $\sigma \rightarrow \langle \text{if } b \text{ then } c, \text{while } b \text{ do } c \text{ else skip}, \sigma \rangle$
Review

- A redex is something that can be reduced in one step
  - E.g. 2+8
- Local reduction rules reduce these redexes
  - E.g. <2+8, σ> → <10, σ>
- Next: global reduction rules
- Consider
  - <x := 1+(2+8), σ>
  - while false do x := 1+(2+8), σ>
- Should we also reduce 2+8 in these cases?

Contexts

- A context is an expression or command with exactly one marker "•".
  - The marker is sometimes called a hole.
  - H[e] is obtained from H by replacing the marker • with e
- Examples
  - x := 1•
    - Fill context H with 2+8 to yield H[2+8] = x := 1(2+8)
    - Or fill context with 10 to yield H[10] = x := 1+10
  - while false do x := 1•
    - Fill with 2+8 to yield H[2+8] = while false do x := 1(2+8)
    - while false do •
- •

Evaluation Contexts

- An evaluation context is a context in which the marker indicates the next place for evaluation.
  - identifies the next redex, a bit like a program counter
  \[ H ::= *
  \begin{cases}
  \text{H + e} \\
  \text{n + H} \\
  \text{x := H} \\
  \text{if H then c1 else c2} \\
  \text{H; c}
  \end{cases} \]
- Examples
  - x := 1•
    - •
    - NOT: while false do x := 1•
    - NOT: if b then c else •

Contexts: Notes

- Evaluation contexts say how to find the next redex:
  - Consider e1 + e2 and its decomposition as H[r].
  - If e1 is n1 and e2 is n2
    - then H = • and r = n1 + n2.
  - If e1 is n1 and e2 is not n2
    - then H = n1 + H2 and e2 = H2[r].
  - If e1 is not n1
    - then H = n1 + e2 and e1 = H2[r].
  - In the last two cases the decomposition is done recursively.
  - In each case the solution is unique.

The Global Reduction Rule

- General idea of the contextual semantics:
  - Decompose the current expression into
    - the next redex r
    - and an evaluation context H (the remaining program)
  - Reduce the redex "r" to some other expression "e".
  - Put "e" back into the original context, yielding H[e].
- Formalized as a small step rule:
  \[ \text{If } \langle r, σ \rangle \rightarrow \langle e, σ' \rangle \text{ then } \langle H[r], σ' \rangle \rightarrow \langle H[e], σ' \rangle \]
The Global Reduction Rule: Example

- Consider the command \( x := 1 + (2 + 8) \)
- Split into an evaluation context \( H \) and a redex \( r \)
- Get
  \[ H = x := 1 + \]
  \[ r = 2 + 8 \]
- \( H \{ r \} = x := 1 + (2 + 8) \) (original command)
- Have
  - \( (2 + 8, \sigma) \rightarrow (10, \sigma) \) (local reduction rule)
- Define global reduction
  - \( (H \{ 2 + 8 \}, \sigma) \rightarrow (H \{ 10 \}, \sigma) \) or, equivalently
  - \( (x := 1 + (2 + 8), \sigma) \rightarrow (x := 1 + 10, \sigma) \)

Contextual Semantics: Example

- Consider the small-step evaluation of \( x := 1 ; x := x + 1 \) in the initial state \( [x := 0] \)

<table>
<thead>
<tr>
<th>State</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1 ; x := x + 1, [x := 0] )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( ; x := x + 1 )</td>
<td>( x := 1 )</td>
<td></td>
</tr>
<tr>
<td>( \text{skip}; x := x + 1, [x := 1] )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( ; x := x + 1 )</td>
<td>( \text{skip}; x := x + 1 )</td>
<td></td>
</tr>
<tr>
<td>( x := x + 1, [x := 1] )</td>
<td>( x := * + 1 )</td>
<td></td>
</tr>
<tr>
<td>( x := 1 + 1, [x := 1] )</td>
<td>( x := * )</td>
<td>1 + 1</td>
</tr>
<tr>
<td>( x := 2, [x := 1] )</td>
<td>( * )</td>
<td></td>
</tr>
<tr>
<td>( \text{skip}, [x := 2] )</td>
<td>( x := 2 )</td>
<td></td>
</tr>
</tbody>
</table>

Normal vs Short-Circuit Boolean Operators

- What if we want normal evaluation of \( \land \) ?
  - Define the following contexts, redexes, and local rules:
    \[ H ::= \ldots | H \land b_2 | p_1 \land H \]
    \[ r ::= \ldots | b_2 | p_2 \]
    \[ \langle p_1 \land p_2, \sigma \rangle \rightarrow \langle \text{true}, \sigma \rangle \]
    \[ \langle \text{false} \land b_2, \sigma \rangle \rightarrow \langle \text{false}, \sigma \rangle \]

- What if we want short-circuit evaluation of \( \land \) ?
  - Define the following contexts, redexes, and local rules:
    \[ H ::= \ldots | H \land b_2 \]
    \[ r ::= \ldots | \text{true} \land b_2 | \text{false} \land b_2 \]
    \[ \langle \text{true} \land b_2, \sigma \rangle \rightarrow \langle b_2, \sigma \rangle \]
    \[ \langle \text{false} \land b_2, \sigma \rangle \rightarrow \langle \text{false}, \sigma \rangle \]
  - The local reduction kicks in before \( b_2 \) is evaluated.

Contextual Semantics: Notes

- One can think of the \# as representing the program counter.
- The advancement rules for \# are not trivial:
  - At each step the entire command is decomposed.
  - This makes contextual semantics inefficient to implement directly.

- The major advantage of contextual semantics is that it allows a mix of local and global reduction rules.
  - For IMP we have only local reduction rules: only the redex is reduced.
  - Sometimes it is useful to work on the context too.

Some Further Topics

- Treatment of errors in operational semantics
  - with an explicit "error" result, as in \((3/0) \rightarrow \text{error})
  - with an "error" expression, as in \((3 + \text{error})
  - with "stuck" computations, so \((3/0) \rightarrow r \) for no \( r \)
- Treatment of overflow (see homework 2)
Contextual Semantics: Notes

- For example: \( c = c_1 \cdot c_2 \)
  - either \( c_1 = \text{skip} \) and then \( c = H[\text{skip}; c_2] \) with \( H = \star \)
  - or \( c_1 \neq \text{skip} \) and then \( c_1 = H[r] \);
    so \( c = c_1; c_2 = H[r]; c_2 = H[r] \) where \( H = H; c_2 \)

- For example: \( c = \text{if } b \text{ then } c_1 \text{ else } c_2 \)
  - either \( b = \text{true} \) or \( b = \text{false} \) and then \( c = H[r] \) with \( H = \star \)
  - or \( b \) is not a value and \( b = H[r] \);
    so \( c = H[r] \) where \( H = H \) if \( H \) then \( c_1 \) else \( c_2 \)

- Decomposition theorem: If \( c \) is not "skip" then there exist unique \( H \) and \( r \) such that \( c = H[r] \).

\( \Rightarrow \) Progress and determinism.

Summary of Operational Semantics

- Precise specification of dynamic semantics:
  - order of evaluation (or that it doesn’t matter)
  - error conditions (sometimes implicitly, by rule applicability)
- Simple and abstract (cf. implementations)
  - no low-level details such as stack and memory management, data layout, etc.
- Often not compositional (as for while)
- Basis for some proofs about languages
- Basis for some reasoning about particular programs
- Point of reference for other semantics

Guidelines for the Final Project

The Final Project

- Three kinds
  - small survey of recent work on a relevant topic (individual)
  - programming project
  - research paper
- Team (1-4 people) or individual projects
- Scale
  - 20-40+ hours of work per person
  - short report
  - short presentation

Picking a Project

- You are encouraged to define your own project.
- If you prefer it, I will be happy to assign you a project, but I can’t guarantee that you will be happy with it.

Scale

- I don’t expect very fancy projects.
  - 20-40 hours of work should suffice, unless you are very enthusiastic.
- However, you are welcome to tackle much more ambitious projects.
  - You should structure such a project so that you can show partial results this quarter.
  - Staged development: very often a good idea
Collaboration

• You can do the projects (except for surveys) in groups of 1-4.

• If you have a great project idea, and it looks too big for one person, feel free to recruit help.

• I think that it is easier to do the projects alone, but you are welcome to make your own choices.

Cheating

• Projects should be new and original
  - not a cut-and-paste of your prior work
  - not also fulfilling the requirements of another course
  - not something you have already finished.

• But it is good if you care about the project beyond completion of this course.

• In a group project, you are expected to do your share.
  - You should notify me if others are not doing theirs.

Kinds of Projects

• There are three basic kinds of projects:
  - Implementation
    - of a small language or algorithm.
  - Survey
    - of work in some area of programming languages.
  - Research
    - on programming languages.

• These kinds can be combined to some extent.
• In all cases, you must write a 2-10 page report.

The Implementation Project

• Implement a small language or an algorithm related to language design, e.g.:
  - an interesting type checking algorithm,
  - a type inference algorithm,
  - some other static analysis algorithm, perhaps based on axiomatic semantics,
  - a big/small step operational semantics,
  - a real language that you invented in the past.

• Write a short report on your project (1-2 pages).
• How much code?
  - 100 lines is probably too small; 10,000 is probably too big.
• Writing a program in an interesting language is not in itself sufficient!

The Survey Project

• Pick an area in which you are interested.
• For example:
  - a family of domain-specific languages,
  - prototyping environments for language design,
  - integration of static and dynamic scoping,
  - implementation strategies for polymorphism,
  - concurrency primitives and objects,
  - type inference for object-oriented languages,
  - axiomatic semantics for parallel languages.

• For more ideas, see for example the proceedings of recent POPL, PLDI, or OOPSLA conferences.

The Survey Project (cont.)

• Read well 2-4 papers.
• Read at least superficially 2-4 extra papers.

• Write a short report on what you have learned.
  - What are the basic problems in this area?
  - What are the basic approaches to solving them?
  - What are the main achievements to date?
The Research Project

• Research projects are the hardest.

• There are several sorts of research projects:
  - Design: invent a language, or part of a language.
  - Modeling: try to formalize some interesting aspect of some existing languages.
  - Theory: extend the theory of language design.
  - Implementation - as discussed earlier.

• In all cases, write a report on this work, of whatever length is appropriate.
  - The writing need not be publication-quality, but it should be able to be read easily.

Reports and Presentations

• Project reports are due at the start of the last class on December 1st.

• We will have brief presentations in the last 4 classes:
  - 7 presentations per class
  - About 10-15 minutes each