Simple Imperative Programming Language #1 (SIMPL1)

\[ I \in \text{Identifiers} \]
\[ N \in \text{Numerals} \]
\[ E ::= N \mid I \mid E + E \mid E \times E \mid E - E \mid I ::= E \]
\[ B ::= \text{true} \mid \text{false} \mid B \& B \mid B \text{ or } B \mid \text{not } B \]
\[ C ::= \text{skip} \mid C \mid \{C\} \mid I ::= E \]
\[ \text{if } B \text{ then } C \text{ else } C \\text{fi} \]
\[ \text{while } B \text{ do } C \]

Commands

- **skip** means done evaluating
- When evaluating an assignment, evaluate expression first
- If the expression being assigned is a value, update the memory with the new value for the identifier
- When evaluating a sequence, work on the first command in the sequence first
- If the first command evaluates to a new memory (ie completes), evaluate remainder with new memory

Transition Semantics

- Aka "small step semantics" or "Structured Operational Semantics"
- Defines a relation of "one step" of computation, instead of complete evaluation
- Determines granularity of atomic computations
- Typically have two kinds of "result": configurations and final values
- Written \((C,m) \rightarrow (C',m')\) or \((C,m) \rightarrow m'\)
Block Command

- Choice of level of granularity:
  - Choice 1: Open a block is a unit of work
    \[ (\{C\}, m) \rightarrow (C, m) \]
  - Choice 2: Blocks are syntactic sugar
    \[ (C, m) \rightarrow (C', m') \]
    \[ (\{C\}, m) \rightarrow (C', m') \]
    \[ (C, m) \rightarrow m' \]
    \[ (\{C\}, m) \rightarrow m' \]

If Then Else Command - in English

- If the boolean guard in an if_then_else is true, then evaluate the first branch
- If it is false, evaluate the second branch
- If the boolean guard is not a value, then start by evaluating it first.

If Then Else Command

\[ (\text{if true then } C \text{ else } C', m) \rightarrow (C, m) \]
\[ (\text{if false then } C \text{ else } C', m) \rightarrow (C', m) \]
\[ (B, m) \rightarrow (B', m) \]
\[ (\text{if } B \text{ then } C \text{ else } C', m) \rightarrow (\text{if } B' \text{ then } C \text{ else } C', m) \]

Example

\[ (y := i; \text{ while } i > 0 \text{ do } \{i := i - 1; y := y * i\}, \langle i \mapsto 3 \rangle) \rightarrow ? \]

Alternate Semantics for SIMPL1

- Can mix Natural Semantics with Transition Semantics to get larger atomic computations
- Use \( (E, m) \Downarrow v \) and \( (B, m) \Downarrow b \) for arithmetics and boolean expressions
- Revise rules for commands
Revised Rules for SIMPL1

Skip: \[(\text{skip}, m) \rightarrow m\]

Assignment: \[\begin{array}{l}
(E, m) \downarrow v \\
(I ::= E, m) \rightarrow m[I ← V ]
\end{array}\]

Sequencing:
- \[(C, m) \rightarrow (C', m')\]
- \[(C, C', m) \rightarrow (C', C', m')\]
- \[(C, m) \rightarrow m'\]

Blocks:
- \[\{C, m\} \rightarrow \{C', m'\}\]
- \[\{C, m\} \rightarrow m'\]

While Command

\[\begin{array}{l}
(B, m) \downarrow \text{true} \\
(\text{while } B \text{ do } C, m) \rightarrow (C, m)
\end{array}\]

\[\begin{array}{l}
(B, m) \downarrow \text{false} \\
(\text{while } B \text{ do } C, m) \rightarrow m
\end{array}\]

\[\text{Other more fine grained options exist (eg rule given before)}\]

Transition Semantics for SIMPL2?

- What are the choices and consequences for giving a transition semantics for the Simple Concurrent Imperative Programming Language #2, SIMPL2?
- For finest grain transitions, summary:
  - Each rule for arithmetic or boolean expression must propagate changes to memory; instead of transitioning to a value, go to a value - memory pair

Transition Semantics for SIMPL2

- Second assignment rule returns value:
  \[(I ::= V, m) \rightarrow (V, m[I ← V ])\]

- Expressions as commands need two rules:
  \[\begin{array}{l}
  (E, m) \rightarrow (E', m') \\
  (E, m) \rightarrow (V, m') \\
  (E, m) \rightarrow m'
  \end{array}\]

Exp. as Comm.: \[\begin{array}{l}
(E, m) \rightarrow (E', m') \\
(E, m) \rightarrow (E', m)
\end{array}\]

Simple Concurrent Imperative Programming Language (SCIMP1)

\[\begin{array}{l}
I \in \text{Identifiers} \\
N \in \text{Numerals} \\
E ::= N \mid I \mid E + E \mid E * E \mid E - E \mid E < E \\
B ::= \text{true} \mid \text{false} \mid B \& B \mid B \| B \mid \text{not } B \\
C ::= \text{skip} \mid C \mid (C) \mid I ::= E \mid C \uparrow C' \\
& \mid \text{if } B \text{ then } C \text{ else } C \text{ fi} \\
& \mid \text{while } B \text{ do } C
\end{array}\]
Semantics for $\parallel$

- $C_1 \parallel C_2$ means that the actions of $C_1$ and done at the same time as, “in parallel” with, those of $C_2$
- True parallelism hard to model; must handle collisions on resources
  - What is the meaning of $x := 1| x := 0$
- True parallelism exists in real world, so important to model correctly

Interleaving Semantics

- Weaker alternative: interleaving semantics
- Each process gets a turn to commit some atomic steps; no preset order of turns, no preset number of actions
- No collision for $x := 1$ and $x := 0$; no collision
- No simultaneous substitution: $x := y| y := x$ results in $x$ and $y$ having the same value; not in swapping their values.

Coarse-Grained Interleaving Semantics for SCIMPL1 Commands

- Skip, Assignment, Sequencing, Blocks, If, Then, Else, While unchanged
- Need rules for $\parallel$

Informal Semantics of sync

- sync($E$) evaluates $E$ to a value $v$
- Waits for another parallel command waiting to synchronize on $v$
- When two parallel commands are both waiting to synchronize on a value $v$, they may both stop waiting, move past the synchronization, and carry on with whatever commands they each have left
- Only two processes may synchronize at a time (in this version).
- Problem: How to formalize?

Simple Concurrent Imperative Programming Language #2 (SCIMP2)

- $l \in$ Identifiers
- $N \in$ Numerals
- $E ::= N | l | E + E | E \ast E | E \ast E | E - E$
- $B ::= \text{true} | \text{false} | B \& B | B \text{ or } B | \text{not } B$
- $C ::= \text{skip} | C | (C) | l ::= E | C \lnot | \text{sync}(E)$
- if $B$ then $C$ else $C$ fi
- while $B$ do $C$

Labeled Transition System (LTS)

A labeled transition system (LTS) is a 4-tuple $(Q, \Sigma, \delta, I)$ where
- $Q$ set of states
  - $Q$ finite or countably infinite
- $\Sigma$ set of labels (aka actions)
  - $\Sigma$ finite or countably infinite
- $\delta \subseteq Q \times \Sigma \times Q$ transition relation
- $I \subseteq Q$ initial states

Note: Write $q \xrightarrow{\alpha} q'$ for $(q, \alpha, q') \in \delta.$
Example: Candy Machine

- $Q = \{\text{Start, Select, GetMarsBar, GetKitKatBar}\}$
- $I = \{\text{Start}\}$
- $\Sigma = \{\text{Pay, ChooseMarsBar, ChooseKitKatBar, TakeCandy}\}$
  
  \[
  \delta = \begin{cases} 
  (\text{Start, Pay, Select}) \\
  (\text{Select, ChooseMarsBar, GetMarsBar}) \\
  (\text{Select, ChooseKitKatBar, GetKitKatBar}) \\
  (\text{GetMarsBar, TakeCandy, Start}) \\
  (\text{GetKitKatBar, TakeCandy, Start}) 
  \end{cases}
  \]