CS 565

Programming Languages (graduate)

Spring 2025

Week 6

A Core Language - IMP

Defining a Language

One “recipe” for defining a language:

1. Syntax:

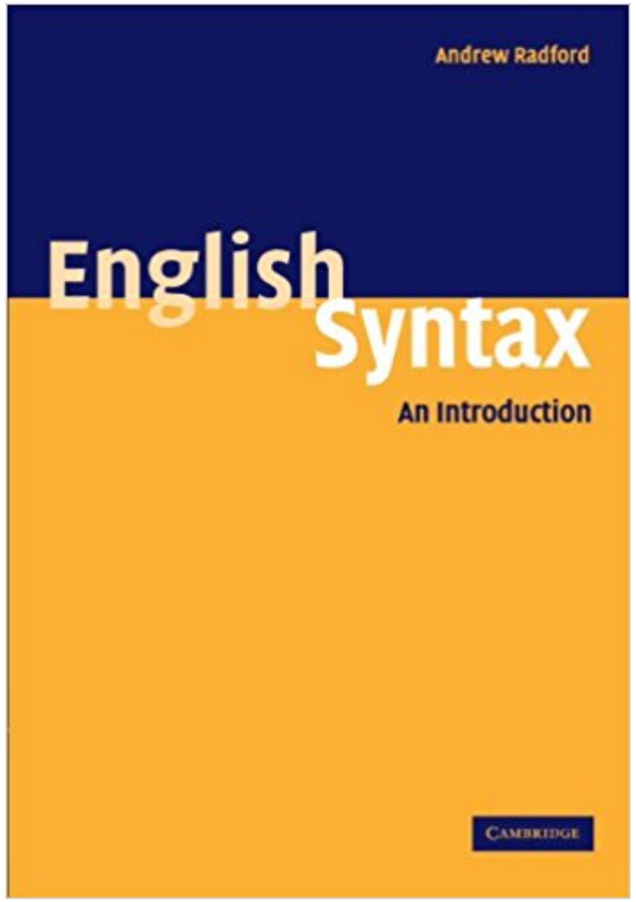
What are the valid sentences?

2. Semantics (Dynamic Semantics):

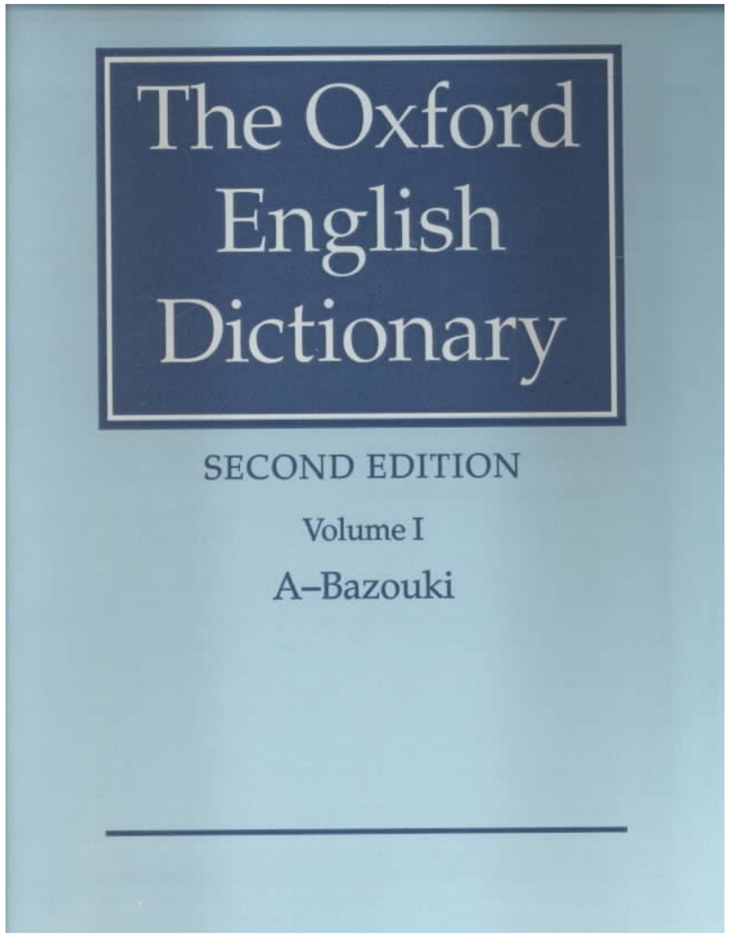
How do I evaluate valid sentences?

Defining English

1. <u>Syntax</u>:



2. <u>Semantics</u>



Defining Haskell

1. Syntax:

topdecl → type simpletype = type
| data [context =>] simpletype [= constrs] [deriving]
| newtype [context =>] simpletype = newconstr [deriving]
| class [scontext =>] tycls tyvar [where cdecls]
| instance [scontext =>] qtycls inst [where idecls]
| default ($type_1$, … , $type_n$) ($n \geq 0$)
| foreign fdecl
| decl

decls → { $decl_1$; … ; $decl_n$ } ($n \geq 0$)
decl → gendecl
| (funlhs | pat) rhs

…

2. Semantics

$$
\begin{array}{lll}
(\Lambda a.e)\ \varphi & \longrightarrow & [\varphi/a]e \\
(\lambda x.e)\ e' & \longrightarrow & [e'/x]e \\
\mathbf{case}\ (K\ \overline{\sigma}\ \overline{\varphi}\ \overline{e})\ \mathbf{of}\ \ldots\ K\ \overline{b}\ \overline{x} \to e'\ \ldots & \longrightarrow & [\overline{\varphi/b}, \overline{e/x}]e' \\
(v \blacktriangleright \gamma_1) \blacktriangleright \gamma_2 & \longrightarrow & v \blacktriangleright (\gamma_1 \circ \gamma_2) \\
 & & \\
((\Lambda a{:}\kappa.\, e) \blacktriangleright \gamma)\ \varphi & \longrightarrow & (\Lambda a{:}\kappa.\, (e \blacktriangleright \gamma @ a))\ \varphi \\
 & \text{where} & \gamma : (\forall a{:}\kappa.\, \sigma_1) \sim (\forall b{:}\kappa.\, \sigma_2) \\
 & & \\
((\Lambda a{:}\kappa.\, e) \blacktriangleright \gamma)\ \varphi & \longrightarrow & (\Lambda a'{:}\kappa'.\, (([(a' \blacktriangleright \gamma_1)/a]e) \blacktriangleright \gamma_2))\ \varphi \\
 & \text{where} & \gamma : (\kappa \Rightarrow \sigma) \sim (\kappa' \Rightarrow \sigma') \\
 & & \gamma_1 = \mathrm{sym}\ (\mathrm{leftc}\ \gamma) \qquad \text{– coercion for argument} \\
 & & \gamma_2 = \mathrm{rightc}\ \gamma \qquad \text{– coercion for result} \\
 & & \\
((\lambda x.e) \blacktriangleright \gamma)\ e' & \longrightarrow & (\lambda y.([(y \blacktriangleright \gamma_1)/x]e \blacktriangleright \gamma_2))\ e' \\
 & \text{where} & \gamma_1 = \mathrm{sym}\ (\mathrm{right}\ (\mathrm{left}\ \gamma)) \qquad \text{– coercion for argument} \\
 & & \gamma_2 = \mathrm{right}\ \gamma \qquad \text{– coercion for result} \\
 & & \\
\mathbf{case}\ (K\ \overline{\sigma}\ \overline{\varphi}\ \overline{e} \blacktriangleright \gamma)\ \mathbf{of}\ \overline{p \to rhs} & \longrightarrow & \mathbf{case}\ (K\ \overline{\tau}\ \overline{\varphi'}\ \overline{e'})\ \mathbf{of}\ \overline{p \to rhs} \\
 & \text{where} & \gamma : T\ \overline{\sigma} \sim T\ \overline{\tau} \\
 & & K : \forall \overline{a{:}\kappa}.\, \forall \overline{b{:}\iota}.\, \overline{\rho} \to T\ \overline{a}^n \\
 & & \varphi'_i = \begin{cases} \varphi_i \blacktriangleright \theta(v_1 \sim v_2) & \text{if } b_i : v_1 \sim v_2 \\ \varphi_i & \text{otherwise} \end{cases} \\
 & & e'_i = e_i \blacktriangleright \theta(\rho_i) \\
 & & \theta = [\overline{\gamma_i/a_i}, \overline{\varphi_i/b_i}] \\
 & & \gamma_i = \mathrm{right}\ \underbrace{(\mathrm{left} \ldots (\mathrm{left}}_{n-i}\ \gamma))
\end{array}
$$

Syntax

(of Imp Commands)

```
C := skip
  | X := A
  | C ; C
  | if B then C
        else C end
  | while B do C end
```

Imp Program

★ Key Feature: State*

```
C := skip
  | x := A
  | C ; C
  | if B then C
        else C end
  | while B do C end
```

```
X := 5;
Z := X;
Y := 1
```

Imp Program

- Key Feature: Control Flow

```
C := skip
  | x := A
  | C ; C
  | if B then C
        else C end
  | while B do C end
```

```
X := 2;
if (X ≤ 1)
   then Y := 3;
            X := 5 - Y
   else Z := 4
end;
Y := 4
```

Imp Program

- <u>Key Feature</u>: Control Flow

```
C := skip
  | x := A
  | C ; C
  | if B then C
        else C end
  | while B do C end
```

```
X := 2;
Z := Y;
while (0 ≤ X) do
   X := X - 1;
   Y := Y + Z
end
```

Imp Program

★ <u>Key Feature</u>: Control Flow

```
C := skip
  | x := A
  | C ; C
  | if B then C
        else C end
  | while B do C end
```

```
X := 2;
Z := X;
Y := 1;
while (0 ≤ Y) do
   X := X - 1;
   Y := Y + Z
end
```

Semantics


```
Fixpoint aeval (a : aexp) (st : var -> int): int =
  match a with
   | ANum n => n
   | APlus a1 a2 => (aeval a1) + (aeval a2)
   | AMinus a1 a2 => (aeval a1) - (aeval a2)
   | AMult a1 a2 => (aeval a1) * (aeval a2)
   | AId x => st x
end.
```

Semantics


```
Fixpoint ceval (c : com) (st : var -> int) :=
 match c with
  | Skip => st
  | Assn x a => update st x (aeval st a)
  | Seq c1 c2 => let st' = ceval st c1 in ceval st' c2
  | If b c1 c2 => if (beval st b) then ceval st c1
                                  else ceval st c2
  | While b c =>  st (* bogus *)
end.
```

Not so clear what to do here: suppose the while loop does not terminate. Then, our formulation of the semantics as an interpreter won't be well-defined either

Semantics

As a Relation

Key Idea: Define evaluation as a Inductive Relation

aevalR: total_map → A → ℕ → Proposition

- ★ Ternary relation on states, expressions and values
- ★ Read ‘σ, a ⇓ n’ as ‘a evaluates to n in state σ’
- ★ Relation precisely spells out what values program can evaluate to
- ★ Put another way, rules define an ‘abstract machine’ for executing expression

Semantics

As a Relation

Key Idea: Define evaluation as a Inductive Relation ($\Downarrow$)

Inference Rules for $\Downarrow$

$$\frac{}{\sigma, n \Downarrow n}\ \text{ENum}$$

$$\frac{}{\sigma, x \Downarrow \sigma(x)}\ \text{EVar}$$

$$\frac{\sigma, e_n \Downarrow v_n \qquad \sigma, e_m \Downarrow v_m}{\sigma, e_n + e_m \Downarrow v_n +_{\mathbb{N}} v_m}\ \text{EAdd}$$

$$\frac{\sigma, e_n \Downarrow v_n \qquad \sigma, e_m \Downarrow v_m}{\sigma, e_n - e_m \Downarrow v_n -_{\mathbb{N}} v_m}\ \text{ESub}$$

$$\frac{\sigma, e_n \Downarrow v_n \qquad \sigma, e_m \Downarrow v_m}{\sigma, e_n * e_m \Downarrow v_n *_{\mathbb{N}} v_m}\ \text{EMult}$$

Reduction

$$\dfrac{\dfrac{\dfrac{}{\sigma, 5 \Downarrow 5}\,\textsc{ENum} \qquad \dfrac{}{\sigma, 2 \Downarrow 2}\,\textsc{ENum}}{\sigma, 5\text{-}2 \Downarrow 5 -_{\mathbb{N}} 2}\,\textsc{ESub} \qquad \dfrac{}{\sigma, 3 \Downarrow 3}\,\textsc{ENum}}{\sigma, 5\text{-}2\text{+}3 \Downarrow 6}\,\textsc{EAdd}$$

Semantics

$$\text{cevalR}: (\text{Id} \rightarrow \mathbb{N}) \rightarrow C \rightarrow (\text{Id} \rightarrow \mathbb{N}) \rightarrow \text{Proposition}$$

- Ternary relation on initial states, commands and final state
- Read 'σ, $c \Downarrow \sigma'$' as 'when run in initial state σ, c produces (i.e. evaluates to) final state σ''

Operational Semantics

Inference Rules for ⇓ (commands)

$$\frac{}{\sigma,\mathrm{skip} \Downarrow \sigma}\ \text{ESKIP}$$

$$\frac{\sigma,c_1 \Downarrow \sigma_1 \qquad \sigma_1,c_2 \Downarrow \sigma_2}{\sigma,c_1;c_2 \Downarrow \sigma_2}\ \text{ESEQ}$$

$$\frac{\sigma,a \Downarrow v}{\sigma,x:=a \Downarrow [x \mapsto v]\sigma}\ \text{EASSN}$$

Operational Semantics

Inference Rules for ⇓ (commands)

EIFT

$$\frac{\sigma, b \Downarrow \textbf{true} \qquad \sigma, c_1 \Downarrow \sigma_1}{\sigma, \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \Downarrow \sigma_1}$$

EIFF

$$\frac{\sigma, b \Downarrow \textbf{false} \qquad \sigma, c_2 \Downarrow \sigma_1}{\sigma, \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \Downarrow \sigma_1}$$

Semantics

Inference Rules for ⇓ (commands)

EWHILET

$$\frac{\sigma_1, b \Downarrow \textbf{true} \qquad \sigma_1, c \Downarrow \sigma_2 \qquad \sigma_2, \textbf{while}\ b\ \textbf{do}\ c\ \textbf{end} \Downarrow \sigma_3}{\sigma_1, \textbf{while}\ b\ \textbf{do}\ c\ \textbf{end} \Downarrow \sigma_3}$$

EWHILEF

$$\frac{\sigma, b \Downarrow \textbf{false}}{\sigma, \textbf{while}\ b\ \textbf{do}\ c\ \textbf{end} \Downarrow \sigma}$$

Why is this a better formulation than the definition of ceval?

Imp Program

$$\sigma,\ \boxed{\begin{array}{l} X := 5; \\ Z := X; \\ Z := 3; \\ Y := 1 \end{array}}\ \Downarrow [Y \mapsto 1][Z \mapsto 3][Z \mapsto 5][X \mapsto 5]\sigma$$

Imp Program

$$\sigma, \boxed{\begin{array}{l} X := 2; \\ \textbf{if } (X \leq 1) \\ \quad \textbf{then } Y := 3 \\ \quad \textbf{else } Z := 4 \\ \textbf{end} \end{array}} \Downarrow [X \mapsto 2]\sigma$$

Imp Program

σ,

```
X := 2;
Z := Y;
while (0 ≤ X) do
  X := X - 1;
  Y := Y + Z
end
```

$\Downarrow$ $[Y \mapsto \sigma(Y)+\sigma(Y)][X \mapsto 0]$
$[Y \mapsto \sigma(Y)+\sigma(Y)][X \mapsto 1]$
$[Z \mapsto \sigma(Y)][X \mapsto 2]\sigma$

Imp Program

σ,

```
X := 2;
Z := X;
Y := 1;
while (0 ≤ Y) do
   X := X - 1;
   Y := Y + Z
end
```

⇓

Defining IMP

1. Syntax

```
C := skip
   | X := A
   | C ; C
   | if B then C
        else C end
   | while B do C end
```

2. Semantics

$$\frac{}{\sigma,\text{skip} \Downarrow \sigma}\ \text{ESKIP}$$

$$\frac{\sigma,a \Downarrow_A v}{\sigma,x{:=}a \Downarrow [x\mapsto v]\sigma}\ \text{EASSN}$$

★**Theorem [IMP IS DETERMINISTIC]:**

For any command c, from any starting state σ, c can evaluate to at most one unique final state:

If $\sigma,\ c \Downarrow \sigma_1$ and $\sigma,\ c \Downarrow \sigma_2$, then $\sigma_1 = \sigma_2$.

Defining IMP +FLIP

1. Syntax

```
C := skip
   | X := A
   | C ; C
   | if B then C
          else C end
   | while B do C end
   | if flip C
```

2. Semantics

$$\frac{\sigma, a \Downarrow v}{\sigma, x{:=}a \Downarrow [x \mapsto v]\sigma} \quad \text{EASSN}$$

$$\frac{\sigma_1, c \Downarrow \sigma_2}{\sigma 1, \texttt{if flip c} \Downarrow \sigma 2} \quad \text{EFLIPT}$$

$$\frac{}{\sigma, \texttt{if flip c} \Downarrow \sigma} \quad \text{EFLIPF}$$

Concept Check

Theorem [IMP+Flip Is Not Deterministic]:

For some commands c, from any starting state σ, c can evaluate to multiple final states:

$\exists\ \sigma\ c\ \sigma_1\ \sigma_2.$ If $\sigma,\ c \Downarrow \sigma_1$ and $\sigma,\ c \Downarrow \sigma_2$ and $\sigma_1 \neq \sigma_2$.

Can you write an IMP+Flip program that evaluates to different final states?

Can you write an IMP+Flip program that evaluates to an infinite number of final states?

Defining IMP +Rand

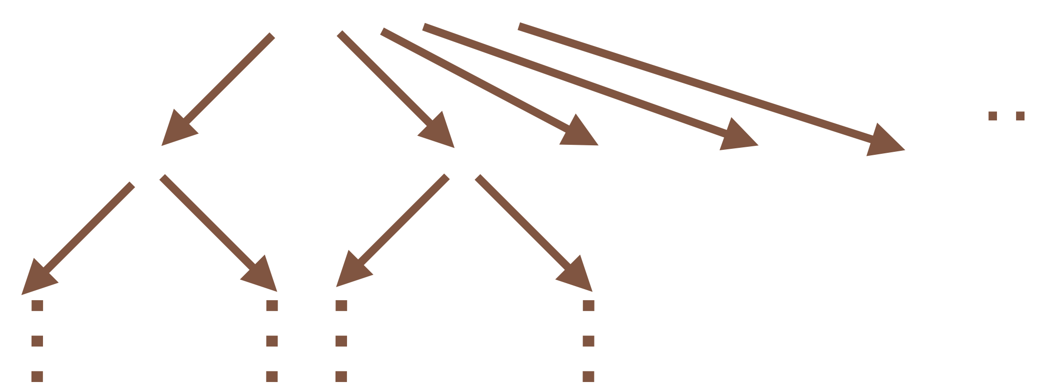
1. Syntax

```
C := skip
   | X := A
   | C ; C
   | if B then C
          else C end
   | while B do C end
   | X := Any
```

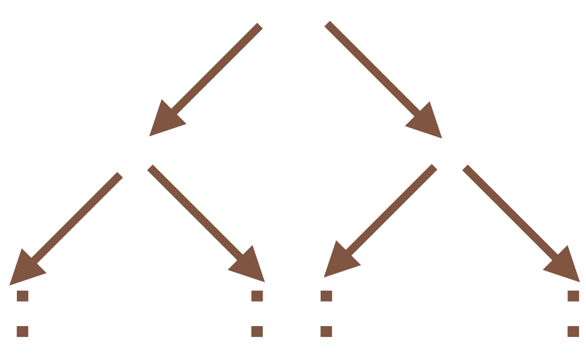
2. Semantics

$$\frac{v \in \mathbb{N}}{\sigma,\ \texttt{x := Any} \Downarrow [x \mapsto v]\sigma}\ \textsc{ERand}$$

IMP+Rand: infinite number of branches
infinite final states



IMP+Flip: finite number of branches
infinite final states



Fun IMP

1. Syntax

C := skip
 | X := A
 | C ; C
 | **if** B **then** C
 else C **end**
 | **while** B **do** C **end**
 | X := F(A)

F $\in \mathbb{F}$

FD := F(X) {C; **return** a}

```
Double(Y) {
 skip;
 return Y + Y}
```

```
Double(Y) {
 Z := Y + Y;
 return Z}
```

```
X := Double(5)
```

```
Y := 5;
X := Double(Y)
```

Fun IMP


```
C := skip
  | X := A
  | C ; C
  | if B then C
        else C end
  | while B do C end
  | X := F(A)

F ∈ 𝔽
FD := F(X) {C; return a}
```

- How to model set of function calls?
- Update the judgement!

$$\Delta \vdash \sigma_1, \ c \Downarrow \sigma_2$$

$$\Delta : F \rightarrow FD$$

Read as 'When run in initial state σ1 and using the function definitions in Δ, c produces (i.e. evaluates to) final state σ2"

Fun IMP

$$\frac{\Delta \vdash \sigma, a \Downarrow v}{\Delta \vdash \sigma, x{:=}a \Downarrow [x \mapsto v]\sigma} \quad \text{EASSN}$$

$$\frac{\Delta(F) = F(y)\ \{c;\ \textbf{return}\ a_2\} \qquad \Delta \vdash \sigma_1, \overline{a} \Downarrow \overline{v} \qquad \Delta \vdash [\overline{y} \mapsto \overline{v}], c \Downarrow \sigma_2 \qquad \Delta \vdash \sigma_2, a_2 \Downarrow v_2}{\Delta \vdash \sigma_1, x{:=}\ F(\overline{a}) \Downarrow [x \mapsto v_2]\sigma} \quad \text{ECALL}$$