formal language (= set of constructs defined by precise rules)

- no functions, objects, records, threads, exceptions
- instead: ints, mutable vars, control flow

abstract syntax

- in common "metalanguage"
- program is statement defined as follows:

$$\begin{array}{lll} s & ::= & \text{skip} \\ & | & x := e \\ & | & s ; s \\ & | & \text{if } e \; s \; s \\ & | & \text{while } e \; s \end{array} \qquad \begin{array}{lll} e & ::= & c \\ & | & x \\ & | & e + e \\ & | & e - e \end{array}$$

$$c \in \mathbb{Z} = \{ \ldots, -1, -2, 0, 1, 2, \ldots \}$$

$$x \in \{ x_1, x_2, \ldots, y_1, y_2, \ldots, z_1, z_2, \ldots \}$$

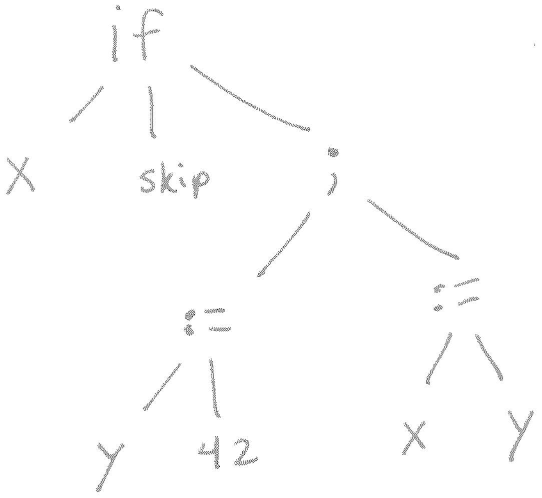
" ::= " means "can be a"

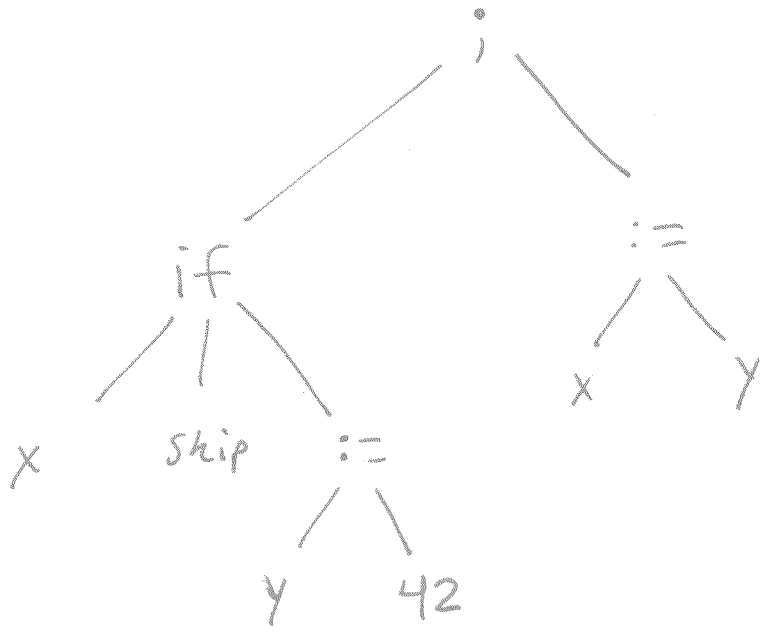
" | " means "or"

Metavariables mean anything in that set $\left(\begin{smallmatrix}\text{syntax}\\\text{class}\end{smallmatrix}\right)$

"abstract syntax" means this defines a set of trees

- nodes have label for constructor ("which alternative")
- children are more ASTs





In OCaml:

```
  type  stmt = Skip
             | Assign of string * exp
             | Seq    of stmt * stmt
             | If     of exp * stmt * stmt
             | While  of exp * stmt

  type  exp  = Const of int
             | Var   of string
             | Add   of exp * exp
             | Mult  of exp * exp

(* pretend ML int = ℤ *)
```

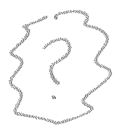
to_str

AST vs. string

Normally write code in concrete syntax, i.e. strings

Potentially ambiguous:

```
if x skip y := 42; x := y
```



However, concrete syntax convenient, so we'll use parens to disambiguate.

Underneath, everything always unambiguous tree.

```
(if x skip y := 42); x := y
```

vs.

```
if x skip (y := 42; x := y)
```

Parsing

parse : string → stmt
(prog)

Heavily studied in 70's & 80's
Fewer papers these days
Lots of good frameworks / tools

Moving forward, always ASTs, but may write as strings for convenience.

Will add parens when necessary to remove ambiguity.

Inductive Definition

our grammar is finite description of ∞ set of trees.

Self reference? Not a problem as long as we use well-founded induction.

- similar to always-terminating recursive function: uses self refence but always "bottoms out"

Precise meaning for our metanotation:

$$E_0 = \emptyset$$

$$(i>0) \quad E_i = E_{i-1} \cup c \cup x \cup \{e_1 + e_2 \mid e_1, e_2 \in E_{i-1}\}$$
$$\cup \{e_1 * e_2 \mid e_1, e_2 \in E_{i-1}\}$$

$$E = \bigcup_{i \geq 0} E_i$$

Our grammar denotes E.

Another (equiv

$$E_0 = c \cup x$$

$$(i > 0) \quad E_i = E_{i-1} \cup \{e_1 + e_2 \mid e_1, e_2 \in E_{i-1}\}$$
$$\{e_1 * e_2 \mid e_1, e_2 \in E_{i-1}\}$$

What is E_1? E_2?

what about S_1? S_2? S_3? S?

Simple Proofs (Proving obvious stuff)

- only have ASTs, but let's get flavor of careful proofs...

Thm: $\exists e$, e has 3 constants

Proof 1: Consider $e = 1 + (2 + 3)$. Need to show $e \in E$.

$E_1 = c \cup x \cup \ldots$ so $1, 2, 3 \in E_1$

$E_2 = \ldots 2+3 \ldots$ so $2 + 3 \in E_2$

$E_3 = \ldots 1 + (2+3) \ldots$ so $1 + (2+3) \in E_3$

$E_3 \subseteq E$ so $e \in E$

finite weight

Proof 2 (Better):

Consider $e = 1 + (2+3)$ and defn of E.

$\boxed{\text{Thm:}}$ $\forall e \in E$, e contains at least 1 const/var.

$\boxed{\text{Proof:}}$ induction on $i \geq 0$

Base: $i = 0 \rightarrow E_i = \emptyset$ vacuous

Ind: Suppose holds for i, show for $i+1$.

$e = c$ ✓
$e = x$ ✓
$e = e_1 + e_2$ IH on e_1 or e_2
$e = e_1 * e_3$ IH on e_1 or e_2

We know e can only be built in certain ways. Exploit in proof.

"Better Proof" (PL-style)

By $\underbrace{\text{"structural induction"}}_{\text{(rules for forming an } e\text{)}}$ on e:

cases:

- ▷ c ✓
- ▷ x ✓
- ▷ $e_1 + e_2$ (assuming holds on e_1, e_2)
- ▷ $e_1 + e_2$ (assuming holds on e_1, e_2)

You get IH on "smaller" terms.
(Just what we were doing w/ i!)

No less precise.
Way more convenient.

exp - ind

for each ctor (X) provide evidence that if prop holds on its args (a1, a2, ..., an) then prop holds on (X a1 a2 ... an)

f
ame
eype

(if X takes no args, must show directly)

Works by replacing nodes in AST w/ sub proofs!

also just this

```
exp-ind =
 fun P : exp -> Prop =>
 fun fc : ∀ n, P (const. n) =>
 fun fv : ∀ s, P (var s) =>
 fun f+ : ∀ e1, P e1 ->
              ∀ e2, P e2 ->
              P (Add e1 e2) =>
 fun f* : ∀ e1, P e1 =>
              ∀ e2, P e2 ->
              P (Mult e1 e2) =>
 fix F e : P e :=
   match e with
   | const n => fc n
   | var s => fv s
   | Add e1 e2 =>
       f+ e1 (F e1) e2 (F e2)
   | Mul => ...
```

Regroup

IMP + structural induction

Next : **Operational Semantics**

So far only said what progs *are*, not what they *mean*.

- We just have a bunch of trees, no notion of how to interpret them.
- We want to precisely capture our intuitions as programmers.

Informally: given exp e,
what does it eval to?

$1 + 2$?

$x + 2$? !!!

Depends on the values of variables!

Need a notion of memory or "heaps".

Use heap H as total func : var → val
- partial gets messy → $\exists\ H, e$ s.t. can't eval

Encode our notion of evaluation (meaning) as a triple over H, e, and c

(Eventually may be a function : $H \times e \to c$)

Heaps

$$\begin{array}{lll} H & ::= & \bullet \\ & \mid & H, x \mapsto c \end{array}$$

lookup :

$$H(x) = \begin{cases} 0 & \text{if } H = \cdot \\ c & \text{if } H = H', x \mapsto c \\ H'(x) & \text{if } H = H', y \mapsto c \land y \neq x \end{cases}$$

"$\bullet$" case forces lookup to be total

- avoids "errors"
- all mem initialized to 0

We will only discuss expr meaning w.r.t. a heap

The Judgement

$\boxed{H ; e \Downarrow c}$ means "e evals to c under H"

just a set of tuples
"true" for tuples in set, false otherwise

- just a relation over $H \times e \times c$, i.e. $\subseteq$
- weird syntax follows PL convention, typical

Consider:

- $\bullet, x \mapsto 3 ; x + y \Downarrow 3$
  - will be true (in our relation)
- $\bullet, x \mapsto 3 ; x + y \Downarrow 6$
  - will be false (not in our relation)