

# CSE-505: Programming Languages

## Lecture 2 — Syntax

Zach Tatlock  
2016

## Finally, some formal PL content

For our first *formal language*, let's leave out functions, objects, records, threads, exceptions, ...

What's left: integers, mutable variables, control-flow

(Abstract) syntax using a common *metalanguage*:

“A program is a statement  $s$ , which is defined as follows”

$$\begin{aligned} s &::= \text{skip} \mid x := e \mid s; s \mid \text{if } e \text{ } s \text{ } s \mid \text{while } e \text{ } s \\ e &::= c \mid x \mid e + e \mid e * e \\ (c &\in \{ \dots, -2, -1, 0, 1, 2, \dots \}) \\ (x &\in \{ x_1, x_2, \dots, y_1, y_2, \dots, z_1, z_2, \dots, \dots \}) \end{aligned}$$

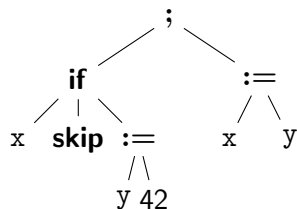
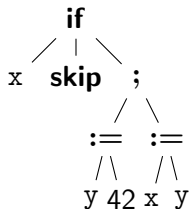
# Syntax Definition

$$\begin{aligned} s &::= \text{skip} \mid x := e \mid s; s \mid \text{if } e \text{ } s \text{ } s \mid \text{while } e \text{ } s \\ e &::= c \mid x \mid e + e \mid e * e \\ (c &\in \{\dots, -2, -1, 0, 1, 2, \dots\}) \\ (x &\in \{x_1, x_2, \dots, y_1, y_2, \dots, z_1, z_2, \dots, \dots\}) \end{aligned}$$

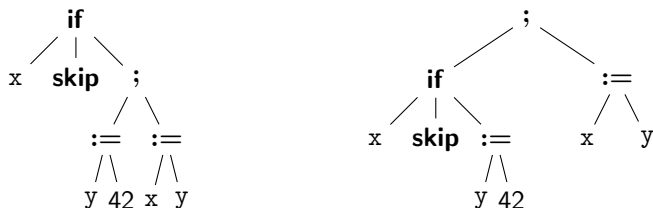
- ▶ Blue is metanotation:  $::=$  for “can be a” and  $|$  for “or”
- ▶ *Metavariables* represent “anything in the *syntax class*”
- ▶ By *abstract syntax*, we mean that this defines a set of *trees*
  - ▶ Node has some label for “which alternative”
  - ▶ Children are more abstract syntax (subtrees) from the appropriate syntax class

# Examples

$s ::= \text{skip} \mid x := e \mid s; s \mid \text{if } e \text{ } s \text{ } s \mid \text{while } e \text{ } s$   
 $e ::= c \mid x \mid e + e \mid e * e$



## Comparison to ML



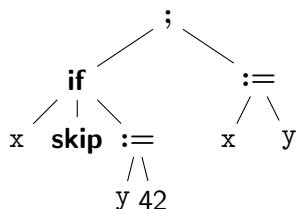
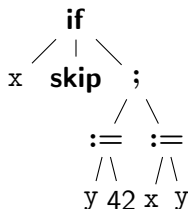
```
type exp = Const of int | Var of string
         | Add of exp * exp | Mult of exp * exp
type stmt = Skip | Assign of string * exp | Seq of stmt * stmt
         | If of exp * stmt * stmt | While of exp * stmt
```

```
If(Var("x"),Skip,Seq(Assign("y",Const 42),Assign("x",Var "y")))
Seq(If(Var("x"),Skip,Assign("y",Const 42)),Assign("x",Var "y"))
```

Very similar to trees built with ML datatypes

- ▶ ML needs “extra nodes” for, e.g., “*e* can be a *c*”
- ▶ Also pretending ML’s `int` is an integer

## Comparison to strings



We are used to writing programs in *concrete syntax*, i.e., strings

That can be *ambiguous*: **if** x **skip** y **:= 42** ; x := y

Since writing strings is such a convenient way to represent trees, we allow ourselves parentheses (or defaults) for disambiguation

- ▶ Trees are our “truth” with strings as a “convenient notation”

**if** x **skip** (y := 42 ; x := y) versus (**if** x **skip** y := 42) ; x := y

## Last word on concrete syntax

Converting a string into a tree is *parsing*

Creating concrete syntax such that parsing is unambiguous is one challenge of *grammar design*

- ▶ Always trivial if you require enough parentheses or keywords
  - ▶ Extreme case: LISP, 1960s; Scheme, 1970s
  - ▶ Extreme case: XML, 1990s
- ▶ Very well studied in 1970s and 1980s, now typically the least interesting part of a compilers course

For the rest of this course, we start with abstract syntax

- ▶ Using strings only as a convenient shorthand and asking if it's ever unclear what tree we mean

## Inductive definition

$$\begin{aligned} s & ::= \text{skip} \mid x := e \mid s; s \mid \text{if } e \text{ s } s \mid \text{while } e \text{ s } \\ e & ::= c \mid x \mid e + e \mid e * e \end{aligned}$$

This grammar is a finite description of an infinite set of trees

The apparent self-reference is not a problem, provided the definition uses well-founded induction

- ▶ Just like an always-terminating recursive function uses self-reference but is not a circular definition!

Can give precise meaning to our metanotation & avoid circularity:

- ▶ Let  $E_0 = \emptyset$
- ▶ For  $i > 0$ , let  $E_i$  be  $E_{i-1}$  union “expressions of the form  $c$ ,  $x$ ,  $e_1 + e_2$ , or  $e_1 * e_2$  where  $e_1, e_2 \in E_{i-1}$ ”
- ▶ Let  $E = \bigcup_{i \geq 0} E_i$

The set  $E$  is what we mean by our compact metanotation



## Inductive definition

$$\begin{aligned} s & ::= \text{skip} \mid x := e \mid s; s \mid \text{if } e \text{ s } s \mid \text{while } e \text{ s} \\ e & ::= c \mid x \mid e + e \mid e * e \end{aligned}$$

- ▶ Let  $E_0 = \emptyset$ .
- ▶ For  $i > 0$ , let  $E_i$  be  $E_{i-1}$  union “expressions of the form  $c$ ,  $x$ ,  $e_1 + e_2$ , or  $e_1 * e_2$  where  $e_1, e_2 \in E_{i-1}$ ”.
- ▶ Let  $E = \bigcup_{i \geq 0} E_i$ .

The set  $E$  is what we mean by our compact metanotation

To get it: What set is  $E_1$ ?  $E_2$ ?

Could explain statements the same way: What is  $S_1$ ?  $S_2$ ?  $S$ ?

## Proving Obvious Stuff

All we have is syntax (sets of abstract-syntax trees), but let's get the idea of proving things carefully...

Theorem 1: There exist expressions with three constants.

# Our First Theorem

There exist expressions with three constants.

Pedantic Proof: Consider  $e = 1 + (2 + 3)$ . Showing  $e \in E_3$  suffices because  $E_3 \subseteq E$ . Showing  $2 + 3 \in E_2$  and  $1 \in E_2$  suffices...

PL-style proof: Consider  $e = 1 + (2 + 3)$  and definition of  $E$ .

Theorem 2: All expressions have at least one constant or variable.

## Our Second Theorem

All expressions have at least one constant or variable.

Pedantic proof: By induction on  $i$ , for all  $e \in \mathbf{E}_i$ ,  $e$  has  $\geq 1$  constant or variable.

- ▶ Base:  $i = 0$  implies  $\mathbf{E}_i = \emptyset$
- ▶ Inductive:  $i > 0$ . Consider *arbitrary*  $e \in \mathbf{E}_i$  by cases:
  - ▶  $e \in \mathbf{E}_{i-1} \dots$
  - ▶  $e = c \dots$
  - ▶  $e = x \dots$
  - ▶  $e = e_1 + e_2$  where  $e_1, e_2 \in \mathbf{E}_{i-1} \dots$
  - ▶  $e = e_1 * e_2$  where  $e_1, e_2 \in \mathbf{E}_{i-1} \dots$

## A “Better” Proof

All expressions have at least one constant or variable.

PL-style proof: By *structural induction* on (rules for forming an expression)  $e$ . Cases:

- ▶  $c \dots$
- ▶  $x \dots$
- ▶  $e_1 + e_2 \dots$
- ▶  $e_1 * e_2 \dots$

Structural induction invokes the induction hypothesis on *smaller* terms. It is equivalent to the pedantic proof, and more convenient in PL