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”

$$
\begin{align*}
  s &::= \operatorname{skip} \mid x := e \mid s_1 ; s_2 \mid \operatorname{if} e \ s_1 \ s_2 \mid \operatorname{while} e \ s \\
  e &::= c \mid x \mid e + e \mid e \ast e
\end{align*}
$$

$(c \in \{\ldots, -2, -1, 0, 1, 2, \ldots\})$

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

Syntax Definition

$s ::= \operatorname{skip} \mid x := e \mid s_1 ; s_2 \mid \operatorname{if} e \ s_1 \ s_2 \mid \operatorname{while} e \ s$

$e ::= c \mid x \mid e + e \mid e \ast e$

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 $y := 42$) ; $x := y$

Comparison to ML

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

Boyana Norris CIS 624 2017, Lecture 2 5

Examples

$$
\begin{align*}
  s &::= \operatorname{skip} \mid x := e \mid s_1 ; s_2 \mid \operatorname{if} e \ s_1 \ s_2 \mid \operatorname{while} e \ s \\
  e &::= c \mid x \mid e + e \mid e \ast e
\end{align*}
$$

$s ::= \operatorname{skip} \mid x := e \mid s_1 ; s_2 \mid \operatorname{if} e \ s_1 \ s_2 \mid \operatorname{while} e \ s$

$e ::= c \mid x \mid e + e \mid e \ast e$

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

Review of Mathematical Induction

A proof by induction that the property \( P(n) \) holds for \( n \in \mathbb{N} \) involves these steps:

- Prove directly that \( P \) is correct for the initial value of \( n \) (for most examples you will see this is zero or one). This is called the base case.
- Assume for some value \( k \) that \( P(k) \) is correct. This is called the induction hypothesis (IH). We will now prove directly that \( P(k) \Rightarrow P(k+1) \). That means prove directly that \( P(k+1) \) is correct by using the fact that \( P(k) \) is correct. This is called the induction step.

Our First Theorem

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

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 $i > 0$, for all $e \in E_i$, $e$ has $\geq 1$ constant or variable.

- Base: $i = 1$ implies $E_i = c, x$, which has at least one constant or variable.
- Inductive: $i > 1$. Consider arbitrary $e \in E_i$ by cases:
  - $e \in E_{i-1}$ ...
  - $e = c$ ...
  - $e = x$ ...
  - $e = e_1 + e_2$ where $e_1, e_2 \in E_{i-1}$ ...
  - $e = e_1 \ast e_2$ where $e_1, e_2 \in E_{i-1}$ ...

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:

- $c$ ...
- $x$ ...
- $e_1 + e_2$ ...
- $e_1 \ast e_2$ ...

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