Programming Language Theory

CSCI-740 Section 01
Prof. Matthew Fluet

MoWeFr 1:25pm – 2:15pm
GAN-2050 / Synchronous Zoom

COVID-19 Reminders:
- Wear a mask (covering nose and mouth)
- Choose an available seat (spread out, obey “do not sit” designations)
  - Seat chosen today will be assigned to you for remainder of semester
- Complete location check-in (QR code)
- No food or drink during class
Programming Language Theory

Abstract Syntax
Today

- Abstract syntax
**IMP**: our first language

Our first *formal language* is as simple as possible.

Leave out:

functions, objects, records, threads, exceptions, . . .

What’s left:

integers, arithmetic, assignment (mutation), (local) control-flow

Abstract syntax using a common *metalanguage*:

“An **IMP** program is a statement $s$, which is defined as follows:”

\[
\begin{align*}
  s &::= x := e \mid \text{skip} \mid s \mid \text{if } e \text{ s s} \mid \text{while } e \text{ s} \\
  e &::= c \mid x \mid e + e \mid e * e \\
  (c &\in \{ \ldots, -2, -1, 0, 1, 2, \ldots \}) \\
  (x &\in \{ x_1, x_2, \ldots, y_1, y_2, \ldots, z_1, z_2, \ldots, \ldots \})
\end{align*}
\]
Syntax definition

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

- Backus-Naur Form (BNF) is the standard metalanguage for syntax
- Blue is metanotation ( ::= “is a”, | “or”)
- Metavariabes (s, e, x, c) represent “anything in the syntax class”
- Symbols ( :=, skip, ...) differentiate alternatives
- But, what have we defined?
Syntax definition

But, *what* have we defined?

- **Concrete syntax:** sequences of symbols
  - ambiguous
    \[
    \text{if } x \text{ skip } y := 0 ; z := 1
    \]

- **Abstract syntax:** trees of labeled nodes and ordered children
  - unambiguous

- use **parentheses** (more *metanotation*) to *disambiguate*
  \[
  \text{if } x \text{ skip } (y := 0 ; z := 1) \quad (\text{if } x \text{ skip } y := 0) ; z := 1
  \]
Why ignore concrete syntax?

- Parsing programming languages is a computer-science success story
- “Solved problem”: take Compiler Construction
- “Boring”:
  - “If it doesn’t work (efficiently), add more keywords/parentheses”
  - Extreeme: put parentheses around everything and don’t use infix
    - 1950s: LISP
    - 1990s: XML

The truth is in the trees!

Assume we have abstract syntax trees.
Syntax definition

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

Abs. syn.: *an infinite set of* trees of labeled nodes and ordered children

- (all?) PLs have an infinite set of programs

Definition is *recursive* (technically, *inductive*), not *circular*:

- 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 BNF metanotation for expressions.

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

Could explain statements the same way. What is $S_1$? $S_2$?
Syntax definition

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

Abs. syn.: *an infinite set of* trees of labeled nodes and ordered children

BNF metanotation provides a finite description of an infinite set.

A Standard ML datatype also provides a finite description of an infinite set:

```ml
datatype exp = Num of int
  | Var of string
  | Plus of exp * exp
  | Times of exp * exp

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

Matthew Fluet
Programming Language Theory
Lecture 02 9
Proving properties of ASTs

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

Theorem 1: There exists an expression with three constants.

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

Theorem 1: There exists an expression 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 $1 \in E_1$ suffices because $E_2$ includes expressions of the form $e_1 + e_2$ where $e_1, e_2 \in E_1$. . . .

PL-style Proof: Consider $e = 1 + (2 + 3)$ and the definition of $e$. 
Our Second Theorem

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

Pedantic Proof: By induction on \( i \), show that for all \( e \in E_i \), \( e \) has \( \geq 1 \) constants or variables.

- **Base Case:** \( i = 0 \) implies \( E_i = \emptyset \). Vacuously true.
- **Inductive Case:** \( i > 0 \). Consider arbitrary \( e \in E_i \) by cases:
  - \( e \in E_{i-1} \): Use induction hypothesis.
  - \( e = c \): True, because \( e \) is one constant.
  - \( e = x \): True, because \( x \) is one variable.
  - \( e = e_1 + e_2 \) where \( e_1, e_2 \in E_{i-1} \). By applying the induction hypothesis to \( e_1 \in E_{i-1} \), we know that \( e_1 \) has \( \geq 1 \) constants or variables. Therefore, \( e \) has \( \geq 1 \) constants or variables.
  - \( e = e_1 * e_2 \) where \( e_1, e_2 \in E_{i-1} \). By applying the induction hypothesis to \( e_1 \in E_{i-1} \), we know that \( e_1 \) has \( \geq 1 \) constants or variables. Therefore, \( e \) has \( \geq 1 \) constants or variables.
Our Second Theorem

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

PL Proof: By *structural* induction on (rules for forming an expression) \( e \).
Consider \( e \) by cases:

- \( e = c \): True, because \( c \) is one constant.
- \( e = x \): True, because \( x \) is one variable.
- \( e = e_1 + e_2 \): By applying the induction hypothesis to \( e_1 \) (an expression smaller than \( e \)), we know that \( e_1 \) has \( \geq 1 \) constants or variables. Therefore, \( e \) has \( \geq 1 \) constants or variables.
- \( e = e_1 \cdot e_2 \): By applying the induction hypothesis to \( e_1 \) (an expression smaller than \( e \)), we know that \( e_1 \) has \( \geq 1 \) constants or variables. Therefore, \( e \) has \( \geq 1 \) constants or variables.

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