Recognizers accept *syntactically correct* programs, but other properties of code *statically* (compile-time) detectable:

- Undeclared variables
- Unused variables
- Compatibility of function type and arguments
- ...

Too complex to be specified by grammars, or recognized by parsers
Static Analysis

Unfortunate but popular terminology: “semantic analysis”

- Formally, *semantics* are dynamic (run-time) properties
- Terminology confuses compile and run-time
- Some resolve confusion with the terminology *static semantics* (vs. *dynamic semantics*)
Type Systems

Static type analysis is a particular form of static analysis:

- Built on rigorous mathematical foundations
- Long history in theoretical computer science (back to when that was the only computer science)
- Various levels of expressiveness, various implementation costs

*Dynamic* type checking also available, used in e.g. Lisp (we won’t consider further)
The Appeal of Type Systems

Type analyses have succeeded in application to PLs for some very good reasons:

- Rigorous foundations allow guarantee of program safety via mathematical proofs
- Expressive type terms provide extra level of program description
- Type can be used to improve efficiency in compilation
Prelude to further discussion

In the following, we will consider some simple languages specified by high-level grammars:

- Grammars not necessarily ready for jcup
- Type analyses specified on expressions in language
- Trivial to translate analysis to work on abstract parse trees in implementation

Approach simplifies theoretical development; implementation simple matter of transcription.
A Simple Language

We begin with a simple language of arithmetic and function definition and application:

\[ \begin{align*}
    x, f & \in \text{Var} & \text{identifiers} \\
    n & \in \mathbb{N} & \text{natural numbers} \\
    e & ::= x | n | e + e | e - e | e(e, \ldots, e) & \text{expressions} \\
    e & ::= \text{let } f(x_1, \ldots, x_n) = e \text{ in } e
\end{align*} \]

For example:

\[
    \text{let } \text{add}(x, y) = x + y \text{ in } \text{add}(2, 3)
\]

Note: recursion disallowed for simplicity (easy to add it)
Dynamic Semantics

To formally specify the run-time semantics of our language, we define an \textit{evaluation} relation $\rightarrow^*$. 

Details omitted for brevity, but for example:

\begin{verbatim}
let add(x, y) = x + y in add(2, 3) $\rightarrow^*$ 5
\end{verbatim}

Mathematical specification allows consideration of semantics in analysis of type system (also provides a cross-platform definition of PL behavior)
Going Wrong

Note that some expressions are semantically ill defined:

\[
\text{let } add(x, y) = x + y \text{ in } add + 4
\]

The subexpression \( add + 4 \) makes no sense here:

- In our formal semantics, \( add + 4 \) cannot be evaluated; such expressions said to be \textit{stuck}
- If an expression \( e \) evaluates to a stuck expression, \( e \) is said to \textit{go wrong}

Goal: rule out expressions that go wrong (semantically ill-defined programs)
Core Dumps and *Worse*

Semantically ill-defined programs undesirable on various levels:

- Unpredictable, platform-specific behaviour
- Segmentation faults, core dumps
- Memory leaks
- Significant security breaches
A Menagerie of Wrongness

Many examples of semantic ill-definedness:

- Applying functions to wrong types of args
- Pointer arithmetic
- Dereferencing dangling pointer
- Out-of-bounds array access
- ...

Here we consider a simple example, but the concepts generalize.
Going Right with Types

In short, a static type analysis allows us to reject semantically ill-defined programs.
To define a type analysis for our simple language, we first define a language of type terms:

\[
\tau ::= \text{int} \mid (\tau_1 \times \cdots \times \tau_n) \to \tau \quad \text{types}
\]

We then update our expression language to include type annotations on function parameters:

\[
x, f \in \text{Var} \quad \text{identifiers}
\]
\[
n \in \mathbb{N} \quad \text{natural numbers}
\]
\[
e ::= x \mid n \mid e + e \mid e - e \mid e(e, \ldots, e) \quad \text{expressions}
\]
\[
e ::= \text{let } f(x_1 : \tau_1, \ldots, x_n : \tau_n) = e \text{ in } e
\]

For example:

\[
\text{let } \text{add}(x : \text{int}, y : \text{int}) = x + y \text{ in } \text{add}(2, 3)
\]
The Type Analysis

We will formally specify our type analysis as a *proof system*:

- Promotes simplicity of presentation
- Eases proof of properties

Type analysis later implemented as *type checking*:

- Independence of specification and implementation promotes uniformity of analysis across platforms
- Must prove correspondance of specification and implementation
Type Judgements: Intuition

To assign a type to an expression, we will use the proof system to deduce valid typing assertions, called *judgements*

Idea: to deduce types for expressions, we recursively descend into expressions

Need to keep track of variable type bindings...
Type Environments

\[ \Gamma ::= \emptyset \mid \Gamma; x : \tau \quad \text{type environments} \]

**Definition 1**  
*We define type environment lookup, denoted \( \Gamma(x) \), inductively as follows:*

\[
\begin{align*}
(\Gamma; x : \tau)(x) &= \tau \\
(\Gamma; y : \tau)(x) &= \Gamma(x) \quad x \neq y
\end{align*}
\]

- LIFO structure on environments
- Extended for declarations in scope
- Implemented as *symbol tables* in automated type checking
Type Judgements

Now, we can define the form of type judgements:

\[ J ::= \Gamma \vdash e : \tau \quad \text{type judgements} \]

We say that a judgement \( \Gamma \vdash e : \tau \) is valid iff it can be deduced.

We write \( e : \tau \) iff \( \emptyset \vdash e : \tau \) is valid.

Note: A fringe benefit of type analysis is a “free-variable analysis”; if \( e : \tau \) then \( e \) contains no undeclared variables.

16
Type Deductions

\[ \text{INT} \quad \text{VAR} \quad \text{PLUS} \]
\[ \Gamma(x) = \tau \quad \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \]
\[ \Gamma \vdash n : \text{int} \quad \Gamma \vdash x : \tau \quad \Gamma \vdash e_1 + e_2 : \text{int} \]

\[ \text{MINUS} \]
\[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \]
\[ \Gamma \vdash e_1 - e_2 : \text{int} \]

\[ \text{APP} \]
\[ \Gamma \vdash e : (\tau_1 \cdots \tau_n) \rightarrow \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma \vdash e_n : \tau_n \]
\[ \Gamma \vdash e(e_1, \ldots, e_n) : \tau \]

\[ \text{FUN} \]
\[ \Gamma; x_1 : \tau_1; \ldots; x_n : \tau_n \vdash e_1 : \tau' \quad \Gamma; f : (x_1 : \tau_1 \cdots x_n : \tau_n) \rightarrow \tau' \vdash e_2 : \tau \]
\[ \Gamma \vdash \text{let } f(x_1 : \tau_1, \ldots, x_n : \tau_n) = e_1 \text{ in } e_2 : \tau \]
Example

\[
\frac{(x : \text{int})(x) = \text{int}}{x : \text{int} \vdash x : \text{int}} \quad \text{VAR} \quad \frac{x : \text{int} \vdash 1 : \text{int}}{x : \text{int} \vdash x + 1 : \text{int}} \quad \text{PLUS}
\]

\[
\frac{(\text{incr} : \text{int} \rightarrow \text{int})(\text{incr}) = \text{int} \rightarrow \text{int}}{\text{incr} : \text{int} \rightarrow \text{int} \vdash \text{incr} : \text{int} \rightarrow \text{int}} \quad \text{VAR} \quad \frac{\text{incr} : \text{int} \rightarrow \text{int} \vdash 2 : \text{int}}{\text{incr} : \text{int} \vdash \text{incr}(2) : \text{int}} \quad \text{APP}
\]

\[
\frac{x : \text{int} \vdash x + 1 : \text{int} \quad \text{incr} : \text{int} \rightarrow \text{int} \vdash \text{incr}(2) : \text{int}}{\emptyset \vdash \textbf{let incr}(x : \text{int}) = x + 1 \textbf{ in incr}(2) : \text{int}} \quad \text{FUN}
\]
Type Safety

Given our formal definition of evaluation and type validity, we can establish type safety:

**Theorem 1 (Type Safety)**  If $e : \tau$ then $e$ does not go wrong.

Well-typedness guarantees semantic well-formedness (run-time safety).
Type Checking

So far, we have discussed the specification of type analysis as a proof system.

Implementation must provide typecheck algorithm that computes valid types:

**Theorem 2 (Type Checking Correctness)** We have that \(\text{typecheck}(\Gamma, e)\) returns \(\tau\) iff \(\Gamma \vdash e : \tau\) is valid.

Defining typecheck for our system so far is trivial; all derivations are deterministic.
Type Checking Algorithm

typecheck(\(\Gamma, n\)) = int

typecheck(\(\Gamma, x\)) = \(\Gamma(x)\)

typecheck(\(\Gamma, e_1 + e_2\)) = if

\[
\begin{align*}
\text{typecheck}(\Gamma, e_1) & = \text{int} \\
\text{typecheck}(\Gamma, e_2) & = \text{int}
\end{align*}
\]

then int

else fail


typecheck(\(\Gamma, e_1 - e_2\)) = if

\[
\begin{align*}
\text{typecheck}(\Gamma, e_1) & = \text{int} \\
\text{typecheck}(\Gamma, e_2) & = \text{int}
\end{align*}
\]

then int

else fail
Type Checking Algorithm

typecheck(\(\Gamma, e(e_1, \ldots, e_n)\)) = if
\[\text{typecheck}(\Gamma, e) = (\tau_1 \ast \cdots \ast \tau_n) \rightarrow \tau\]
\[\text{typecheck}(\Gamma, e_1) = \tau_1\]
\[\vdots\]
\[\text{typecheck}(\Gamma, e_n) = \tau_n\]
then \(\tau\)
else fail
Type Checking Algorithm

typecheck(\Gamma, \text{let } f(x_1: \tau_1, \ldots, x_n: \tau_n) = e_1 \text{ in } e_2) =

\text{let } \Gamma' = \Gamma; x_1: \tau_1; \ldots; x_n: \tau_n
\text{let } \tau = \text{typecheck}(\Gamma', e_1)
\text{let } \Gamma'' = \Gamma; f: (\tau_1 \ast \cdots \ast \tau_n) \rightarrow \tau
\text{in } \text{typecheck}(\Gamma'', e_2)
Type Checking Algorithm

Note: this is essentially the core of the Lake type checker you will implement in Assignment 6.

Type checking can be implemented:

- Post-parsing
- During parsing (parsetime type-checking):
  - Type checking in parser actions
  - Type checking during abstract parse tree construction (Assignment 6 approach)

Parsetime checking more efficient, second pass over completed parse trees not required.
Type Reconstruction

So far, our type analysis has required type annotations on function parameters:

- Unwieldy
- Inconvenient

A more sophisticated alternative is *type reconstruction*, aka *type inference*.
Type Reconstruction

Type reconstruction “discovers” a valid type for programs, eliminating the need for type annotations.

To begin, we add type variables to our language of types:

\[
\begin{align*}
  t & \in Tyvar \\
  \tau & ::= t \mid \text{int} \mid (\tau_1 \ast \cdots \ast \tau_n) \rightarrow \tau
\end{align*}
\]

Type judgements are defined as before.
Type Deductions (no type annotations)

\[
\begin{align*}
\text{INT} & \quad \text{VAR} & \quad \text{PLUS} \\
\Gamma(n) = \tau & \quad \Gamma \vdash x : \tau & \quad \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \\
\Gamma \vdash n : \text{int} & \quad \Gamma \vdash x : \tau & \quad \Gamma \vdash e_1 + e_2 : \text{int}
\end{align*}
\]

\[
\begin{align*}
\text{MINUS} \\
\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \\
\Gamma \vdash e_1 - e_2 : \text{int}
\end{align*}
\]

\[
\begin{align*}
\text{APP} \\
\Gamma \vdash e : (\tau_1 \cdots \tau_n) \rightarrow \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma \vdash e_n : \tau_n \\
\Gamma \vdash e(e_1, \ldots, e_n) : \tau
\end{align*}
\]

\[
\begin{align*}
\text{FUN} \\
\Gamma ; x_1 : \tau_1 ; \ldots ; x_n : \tau_n \vdash e_1 : \tau' \quad \Gamma ; f : (x_1 \cdots x_n) \rightarrow \tau' \vdash e_2 : \tau \\
\Gamma \vdash \text{let } f(x_1, \ldots, \tau_n) = e_1 \text{ in } e_2 : \tau
\end{align*}
\]
Type Reconstruction Algorithm

No longer trivial! Note that judgements are not deterministic (how to “guess” function parameter types?)

Too complicated to give details; intuition:

- Recursively descend into expressions, assign type variable $t$ to parameters in environment
- At variable use point, collect type equation expressing constraint imposed by use:

  $x : t \quad x + 1 \quad t = \text{int}$

- After collecting all such equations in a set $E$, use unification to solve equations
- Solution is a substitution that makes equations true, use to generate inferred types
Other Extensions

Type systems can get still fancier:

- **Polymorphism**— allows program objects to assume multiple types, e.g. stacks which are abstract wrt their contents

- **Subtyping**— allows program objects to assume “supertypes”, e.g. int can be float, OO objects can assume types of superclass
  - Note: *not* the same as coercion

What’s the point?
Conservative Approximations

Type systems can only be so precise: some operationally safe programs rejected:

\[
\text{if } b \text{ then 1 else true}
\]

Type systems are a conservative approximations of run-time behavior: extensions relax conservatism.

Trick of type analysis design: be as flexible as possible, without making analysis impractical.
New Horizons

Type theorists are also seeking to use types for other applications:

- Statically enforce PL security
  - Type safety can be extended to languages with *access control* features

- Type-directed compiler optimizations
  - If type of objects are known, precise space bounds can be allotted
  - Types usually discarded after parse-tree analysis, but...
  - Types-in-compilation research maintains type information throughout compiler transformations

- Applications in natural language processing

- Theorem provers

- ...