Compiler Phases

Source Code (concrete syntax)

```
if (x == 0) x = x + 1;
```

Token Stream

```
if ( x == 0 ) x = x + 1 ;
```

Abstract Syntax Tree (AST)

```
IF
  ==
  x
  0
  =
  +
  x
  1
```

Attributed AST

```
IF
  boolean
  ==
  x
  int
  0
  int
  x
  +
  x
  int
  1
  int
```

Code Generation

```
16: iload_2
17: ifne 24
20: iload_2
21: iconst_1
22: iadd
23:istore_2
24: ...
```
• Type theory covers a huge range of topics
• Several lectures in the courses
  - Programming Language Concepts (344)
  - Programming Language Theory (740)
• In this course we do not cover the theoretical aspects of type system design
• We are mostly interested in **type checking** as a major component of the semantic analysis phase
What is a type?

- Type: a set of values and a set of operations on those values
- Example: Integers
- `int x, y;` means:
  - `x, y ∈ [−2^{31}, 2^{31})`
  - Operations `+ − < <= mod ...` are possible on `x` and `y`
- Type errors:
  improper, type-inconsistent operations during program execution
- Type safety: absence of type errors at run time
How to Ensure Type-Safety?

Bind (assign) types, then check types

**Type binding**

- Defines types for constructs in the program (e.g., variables, functions)
- Can be either explicit (boolean \( x \)) or implicit (\( x = false \))
- Type safety: correctness with respect to the type bindings

**Type checking**

- Static semantic checks to enforce the type safety of the program
- Enforce a set of type-checking rules
Type Check Examples

- Operators (such as +) receive the right types of operands
- User-defined functions receive the right types of operands
- LHS of an assignment should be “assignable”
- Variables are assigned the expected kinds of values
- Return statement must agree with return type
- Class members accessed appropriately
Static vs. Dynamic Typing

- **Statically** typed language: types are defined and checked at compile-time, and do not change during the execution of the program
- E.g., C, Java, Pascal
- **Dynamically** typed language: types defined and checked at run-time, during program execution
- E.g., Lisp, Scheme, Smalltalk
Why Static Checking?

- Efficient code: dynamic checks slow down the program
- Guarantees that all executions will be safe
- With dynamic checking, you never know when the next execution of the program will fail due to a type error

Drawbacks

- Adds an annotation burden for programmers
- Static type safety is a conservative approximation of the values that may occur during all possible executions
- It may reject some type-safe programs unfairly
We have used the following formal notations for specifying the first two phases of compiler:
- Regular expressions for lexical analysis
- Context-free grammars for parsers

We use inference systems from logic to formalize type checking
- Similar to what we did in name analysis

Inference systems are suitable for performing computations of form:

If the first expression is of type $T$ and the second expression is of type $T'$ then the third expression must be of type $T''$
• Example inference rule:

All great universities have smart students  Premise 1
RIT is a great university  Premise 2
RIT has smart students  Conclusion

• Example inference rule:

$e_1$ has type int  Premise 1
$e_2$ has type int  Premise 2
$e_1 + e_2$ has type int  Conclusion
• An inference system has two parts:
  1. Definition of **Judgments**
     - Judgment: statement asserting a certain fact for an object
  2. Finite set of **Inference Rules**

• An inference rule has:
  1. a finite number of judgments $P_1, P_2, \cdots, P_n$ as premises;
  2. a single judgment $C$ as conclusion

• If a rule has no premises, it is called an **axiom**

\[
\begin{array}{cccc}
P_1 & P_2 & \cdots & P_n \\
\hline
C
\end{array}
\]

(Rule name)  

Premises above the line (0 or more)  

Conclusion below the line
**Example:** Use an inference system to define the set of even numbers

- **Judgment:** $\text{Even}(n)$ asserts that $n$ is an even number
- **Inference rules:**
  - **Axiom:**
    
    $$
    \frac{\text{Even}(0)}{(\text{Even0})}
    $$

  - **Successor Rule:**
    
    $$
    \frac{\text{Even}(n)}{\text{Even}(n + 2)} (\text{EvenS})
    $$
Derivation Tree

\[
\begin{array}{c}
\text{Even}(0) \quad \text{(Even0)} \\
\hline
\text{Even}(n) \\
\hline
\text{Even}(n+2) \quad \text{(EvenS)}
\end{array}
\]

- To derive more judgments we create **trees** of inference rules

\[
\begin{array}{c}
\text{Even}(0) \quad \text{(Even0)} \\
\hline
\text{Even}(2) \quad \text{(EvenS)} \\
\hline
\text{Even}(4) \quad \text{(EvenS)} \\
\hline
\text{Even}(6) \quad \text{(EvenS)}
\end{array}
\]
To derive more judgments we create **trees** of inference rules.

- \( \frac{\text{Even}(0)}{} \) (Even0)
- \( \frac{\text{Even}(n)}{\text{Even}(n + 2)} \) (EvenS)

\[\begin{array}{c}
\frac{\text{Even}(0)}{} \quad \text{(Even0)} \\
\frac{\text{Even}(2)}{} \quad \text{(EvenS)} \\
\frac{\text{Even}(4)}{} \quad \text{(EvenS)} \\
\frac{\text{Even}(6)}{} \quad \text{(EvenS)}
\end{array}\]

- Does \( \text{Even}(1) \) hold?
- No, because there exists no possible derivation.
Derivation Tree

Axioms

Judgment

Judgment

Rules

Judgment

Judgment

Judgment

Judgment

Judgment
Example: Use an inference system to define the less-than relation

- Judgment: $n < m$ asserts that $n$ is smaller than $m$
- Inference rules:
  - Axiom: 
    
    $ \begin{array}{c}
    \frac{}{n < n + 1} \\
    \text{(Suc)}
    \end{array}$
  
  - Transitivity Rule:
    
    $ \begin{array}{c}
    k < n \\
    n < m
    \end{array}$
    
    $k < m$ (Trans)

Exercise: Prove $0 < 3$. 
Type Judgments and Type Rules

- $e$ type checks to $T$ under $\Gamma$ (type environment)
  \[ \Gamma \vdash e : T \]

  - Types of constants are predefined
  - Type binding: types of variables are specified in the source code

- If $e$ is composed of sub-expressions
  \[
  \frac{\Gamma \vdash e_1 : T_1 \quad \cdots \quad \Gamma \vdash e_n : T_n}{\Gamma \vdash e : T}
  \]
Type Judgments and Type Rules

If the (free) variables of $e$ have types given by the type environment gamma, then $e$ (correctly) type checks and has type $T$

\[
\Gamma \vdash e : T
\]

If $e_1$ type checks in $\Gamma$ and has type $T_1$ and ... and $e_n$ type checks in $\Gamma$ and has type $T_n$ then $e$ type checks in $\Gamma$ and has type $T$

\[
\frac{\Gamma \vdash e_1 : T_1 \quad \cdots \quad \Gamma \vdash e_n : T_n}{\Gamma \vdash e : T}
\]
Type Rules with Environment

int x;
int y;

(x < y) ? x : (y + 1)

Type Rules:

\[
\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}
\]

\[\text{IntConst}(k) : \text{int}\]

\[
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash (e_1 < e_2) : \text{boolean}}
\]

\[
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash (e_1 + e_2) : \text{int}}
\]

\[
\frac{\Gamma \vdash b : \text{boolean} \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash (b \ ? \ e_1 : e_2) : T}
\]