Semantic Analysis

- Lexical analysis: From character sequences to token sequences.
- Syntactic analysis: From token sequences to the abstract syntax tree, i.e., a tree-based representation of the structure of the program.
- Next question: Is the structure of the program *meaningful*, i.e., does the program “make sense”?
- This is the domain of semantic analysis. Example questions:
  - [Pascal] Does the declaration of each identifier precede its uses?
  - [All] Does the call of a procedure match its type signature?
  - [Rust] Have all variables been initialized before they are used?
  - [C] Does a `break` statement have an appropriate enclosing construct?
  - [Ada] Does the same name occur both at the beginning and the end of a named loop or block?
  - [APL] What is the type and dimensionality of a name at a program point?

Beyond Context-Free Language Features

- The following formal languages can be proven to not be context-free.
  - $L_1 = \{wcw \mid w \in (a|b)^*\}$.
  - $L_2 = \{a^n b^m c^n d^m \mid n \geq 1 \wedge m \geq 1\}$.

Beyond Context-Free Language Features

- The following formal languages can be proven to not be context-free.
  - $L_1 = \{wcw \mid w \in (a|b)^*\}$.
  - $L_2 = \{a^n b^m c^n d^m \mid n \geq 1 \land m \geq 1\}$.
- These languages abstract semantic analysis problems.
  - L_1 abstracts the problem of checking that identifiers are declared before their use in a program.
  - L_2 abstracts the problem of checking that the number of formal parameters in the declaration of a procedure agrees with the number of actual arguments in a use of the procedure.

Beyond Context-Free Language Features

- The following formal languages can be proven to not be context-free.
  - $L_1 = \{wcw \mid w \in (a|b)^*\}$.
  - $L_2 = \{a^n b^m c^n d^m \mid n \geq 1 \wedge m \geq 1\}$.
- These languages abstract semantic analysis problems.
  - L_1 abstracts the problem of checking that identifiers are declared before their use in a program.
  - L_2 abstracts the problem of checking that the number of formal parameters in the declaration of a procedure agrees with the number of actual arguments in a use of the procedure.
- Why not use context-sensitive grammars?
  - The problem of parsing a context-sensitive grammar is PSPACE-complete.
  - Even a CSG would have difficulty (or outright lack the power) to effective encode typical semantic analysis problems.

Canonical Example: Type Checking

- We will use type checking as our canonical example of a semantic analysis problem.

Canonical Example: Type Checking

- We will use type checking as our canonical example of a semantic analysis problem.
- What is a type?
    - [Cardelli & Wegner 1985] "Types arise informally in any domain to categorize objects according to their usage and behavior. [...] Sets of uniform objects may be named and are referred to as types. [...] A type may be viewed as a set of clothes (or a suit of armor) that protects an underlying untyped representation from arbitrary or unintended use."
    - [Cooper & Torczon 2004] "The type specifies a set of properties held in common by all values of that type. [...] Types can be specified by membership; [t]ypes can be specified by rules."
    - [K&R2e] "... the type determines the meaning of the values found in the identified object."

Canonical Example: Type Checking

- We will use type checking as our canonical example of a semantic analysis problem.
- What is a type?
  - [Cardelli & Wegner 1985] “Types arise informally in any domain to categorize objects according to their usage and behavior. [...] Sets of uniform objects may be named and are referred to as types. [...] A type may be viewed as a set of clothes (or a suit of armor) that protects an underlying untyped representation from arbitrary or unintended use.”
  - [Cooper & Torczon 2004] “The type specifies a set of properties held in common by all values of that type. [...] Types can be specified by membership; [t]ypes can be specified by rules.”
  - [K&R2e] “... the type determines the meaning of the values found in the identified object.”
- The set of types in a programming language, along with the rules that use types to specify program behavior, are collectively called a type system.

The Purpose of Type Systems

- Ensuring run-time safety.
  - Attempt to identify and catch as many ill-defined programs as possible before they execute an operation that causes a run-time error.
  - Infer a type for each expression. Check the types of operands to an operator with the language specification of that operator. Possibly coerce values to fit the specification.

The Purpose of Type Systems

- Ensuring run-time safety.
  - Attempt to identify and catch as many ill-defined programs as possible before they execute an operation that causes a run-time error.
  - Infer a type for each expression. Check the types of operands to an operator with the language specification of that operator. Possibly coerce values to fit the specification.
- Improving expressiveness.
  - Specify behavior more precisely than possible with context-free rules.
  - Operator overloading: what does + signify in most modern procedural languages?

The Purpose of Type Systems

- Ensuring run-time safety.
  - Attempt to identify and catch as many ill-defined programs as possible before they execute an operation that causes a run-time error.
  - Infer a type for each expression. Check the types of operands to an operator with the language specification of that operator. Possibly coerce values to fit the specification.
- Improving expressiveness.
  - Specify behavior more precisely than possible with context-free rules.
  - Operator overloading: what does + signify in most modern procedural languages?
- Generating better code.
  - If the compiler can accurately determine the types of every expression statically, it can generate type-specific assembly code.
  - This avoids the overheads of maintaining run-time data tags (space) and disambiguating types at run-time (time).

Some Terminology

- Languages
  - *Statically typed*: Every expression can be type-checked at compile time.
  - *Dynamically typed*: Some expressions can be type-checked only at run time.
    - *Untyped*: Really, only has one type (e.g., BCPL).
    - *Weakly typed*: Has a poor type system.
  - *Strongly typed*: Every expression can be assigned an unambiguous type.

Some Terminology

- Languages
  - *Statically typed*: Every expression can be type-checked at compile time.
  - *Dynamically typed*: Some expressions can be type-checked only at run time.
    - *Untyped*: Really, only has one type (e.g., BCPL).
    - *Weakly typed*: Has a poor type system.
  - *Strongly typed*: Every expression can be assigned an unambiguous type.
- Type systems and implementations
  - *Strongly checked*: Perform enough checking to detect and prevent all run-time errors that result from misusing a type.
  - *Unchecked*: The implementation assumes a well-formed program.
  - *Statically checked*: All type inference and checking is actually done at compile time.

Some Terminology

- Languages
  - *Statically typed*: Every expression can be type-checked at compile time.
  - *Dynamically typed*: Some expressions can be type-checked only at run time.
    - *Untyped*: Really, only has one type (e.g., BCPL).
    - *Weakly typed*: Has a poor type system.
  - *Strongly typed*: Every expression can be assigned an unambiguous type.
- Type systems and implementations
  - *Strongly checked*: Perform enough checking to detect and prevent all run-time errors that result from misusing a type.
  - *Unchecked*: The implementation assumes a well-formed program.
  - *Statically checked*: All type inference and checking is actually done at compile time.
- Java could be statically typed and checked if the execution model allowed seeing all the code at once.
  - Since this is not the case, type inference must be perform as classes are loaded, and some run-time checking is performed.

Judgments and Typing Rules

- Typing rules contain judgments of the form $E \Rightarrow e: T$, where E is an environment containing, for example, the types of identifiers and functions.
  - Read the judgment as "In the environment E, expression e has type T."

Judgments and Typing Rules

- Typing rules contain judgments of the form $E \Rightarrow e{:}\,T$, where E is an environment containing, for example, the types of identifiers and functions.
  - Read the judgment as "In the environment E, expression e has type T."
- When we need to, we will also explicitly divide the environment E into a signature F and a context G.
  - The signature shows the types of functions.
  - The context shows the types of variables.
  - So we will write $F, G \Rightarrow e{:}\,T$.

Judgments and Typing Rules

- Typing rules contain judgments of the form $E \Rightarrow e: T$, where E is an environment containing, for example, the types of identifiers and functions.
  - Read the judgment as "In the environment E, expression e has type T."
- When we need to, we will also explicitly divide the environment E into a signature F and a context G.
  - The signature shows the types of functions.
  - The context shows the types of variables.
  - So we will write $F, G \Rightarrow e: T$.
- Typing rules have the form

$$\frac{J_1 \quad J_2 \quad \ldots \quad J_n}{J} \; C \; (n \geq 0)$$

  - Read the rule as "From the judgments J_1 through J_n, if condition C holds, conclude J."
  - Judgments expressed in formal language; condition in natural language.

Examples of Typing Rules and Derivations

- Typing rules for arithmetic expressions
  - $\dfrac{E \Rightarrow e_1{:}\mathbf{int} \quad E \Rightarrow e_2{:}\mathbf{int}}{E \Rightarrow e_1{+}e_2{:}\mathbf{int}}, \quad \dfrac{E \Rightarrow e_1{:}\mathbf{int} \quad E \Rightarrow e_2{:}\mathbf{int}}{E \Rightarrow e_1{*}e_2{:}\mathbf{int}}.$
  - $\dfrac{}{E \Rightarrow x{:}T}\ x{:}\,T \in E, \quad \dfrac{}{E \Rightarrow i{:}\mathbf{int}}$ i is an integer literal.

- Given these typing rules, how does one derive the judgment $x{:}\,\mathbf{int}, y{:}\,\mathbf{int} \Rightarrow x + 12 * y{:}\,\mathbf{int}$?