Static Semantic Analysis

Binding and type checking

Syntax or Semantics?

• “Declare before use” is really a syntactic rule, but not context free
  – So we treat it as a “static semantic” check
• Types are (arguably) really semantics
  – Because they capture an invariant of execution: V will always hold a T value
• Why should you care?
  – Occasionally we can choose between checking something in the grammar, or in static semantics; deferring a check may simplify the parser

Some things to check ...

• In coolc
  – All identifiers are declared
  – Types
  – Inheritance relationships
  – Classes are declared only once
  – Methods in a class are defined only once
  – Reserved identifiers are not redefined
  – etc
• Details vary from language to language

Static Semantics

• “The end of the beginning”
  – last of the front-end phases; it’s all downstream from here
  – ought to be the last source of error messages to the user; any downstream errors are bugs in the compiler
• Everything you can’t (or don’t) do in the parser

Parts of this lecture are stolen/adapted from Prof. Alex Aiken’s lecture notes.
Binding (Scope) Analysis

- Checks that identifiers are declared
  - Mostly before they are used
  - And not redefined unless they are allowed
- Both an error-checking stage, and an information-building stage
  - Adding information to the AST and/or a symbol table

Static and Dynamic Scope

- Static: Based on program text
  - Used in most general purpose languages (GPPLs), for most things
    - including C, Java, Cool
- Dynamic: Based on program execution
  - Used in many little languages (e.g., PostScript), and a few GPPLs
  - May have some role in (mostly) statically scoped languages, e.g., exception handling

Lexical Scoping

- A static scope rule: Name refers to the closest *enclosing* definition of that name
  - Popularized in Algol and its descendants (Pascal, Ada, Modula)
  - Some role (but lesser) in C, C++, Java
    - Blocks can be nested (and now classes), but methods/functions cannot be nested.
    - Aside: JVM does not support lexical scope well; that’s why inner classes are such a hack.

Cool Scopes

- Identifier bindings introduced by
  - Class declarations (class names)
  - Method definitions (method names)
  - Let expressions (object names)
  - Formal parameters (object names)
  - Attribute definitions (object names)
  - Case expressions (object names)
Exceptions to Lexical Scope

• Some bindings in Cool do not follow the “most closely nested” lexical scope rule
• Example: Class definitions
  – Cannot be nested
  – Are globally visible (one global scope for all classes)
  – Including visibility before the definition!
    • i.e., you can use class C before you declare it

Global Scope of Classes

Class Foo {
  ... let y: Bar in ...
}
Class Bar {
  ...
}

• Needed for recursive data structures and other recursive reference
• Common alternative is some form of stub or “forward” declaration
  – Prototypes in C, “forward” in Pascal, etc.

Attribute Scope

• Attributes are global within a class
  – i.e., limited scope, but no “definition before use” rule

Class Foo {
  f(): Int { a };
  a: Int <- 0;
}

Binding and Inheritance

• Methods and attributes can be defined in the class in which they are used, or a parent class
• Methods in parent classes may be redefined in subclasses (overriding)
  – Is this like the lexical most-closely-nested rule? How is it similar and different?
Implementing Most-Closely-Nested

• Follows a stack discipline
  – Enter (push) declarations where they are encountered
  – Remove (pop) declarations on scope exit

• Common symbol table organizations
  – Stack of tables
  – Table of stacks
  – Tree with “first found” rule

Aside: Name Spaces

• Some languages (like C) distribute names into name spaces
  – distinct symbol tables; we must be able to tell which to look in from the grammar

• Names in one symbol table (e.g., function names) don’t hide or clash with names in another (e.g., variables)

Limitations of Simple Stack Discipline

• We shouldn’t allow this:
  
  foo (x: Int, x: String);

  Declarations should only cover declarations in other scopes, not this one

Symbol Table with Scopes

• enter_scope( ) start a new nested scope
• find_symbol(x) returns nearest def or null
• check_scope(x) true if x defined in current scope
• exit_scope(x) delete all since enter

Provided module
Class Definitions

• Class names can be used before definition

• So: Neither purely inherited nor synthesized, nor an L-attribution
  – We can’t check class names in one pass; at least two walks of the AST are required