Agenda

- Abstract Syntax Trees
- Identifiers and names
- Symbol tables
Abstract Syntax Trees

- An tree-structured *intermediate representation* of a program that abstracts non-essential syntactic details, while retaining the fundamental shape of the input.

- Generated by the parser, consumed by subsequent analysis phases.

- While parse trees (concrete syntax) follow the specific productions of the grammar, the abstract syntax keeps only the semantically meaningful concepts.
Annotating AST

- ASTs start as trees: single root, directed graph, acyclic

- It’s sometimes useful to annotate the tree with additional graph edges

- Additional book keeping:
  - Type information
  - Symbols
  - Runtime consideration
  - Code generation: offsets, labels, stack memory layout, etc.

- Modify in-place or generate a transformed copy in the process
Backlinks
Other useful links

• From \texttt{break}'s to their enclosing loop

• From \texttt{return}'s to their enclosing function

• From \texttt{else} arms to the initial \texttt{if} or to sibling \texttt{else}'s

• From a nested function to their enclosing parent function
  • Nested scopes

• From identifier \textit{uses} to their \textit{declarations}
Identifier uses & declarations

```
Declare Ident "x" Type "integer"
AssignStmt Ident "x" Const 0
```
From *identifiers* to *symbols*

```
Declare Ident "x" Type "integer"
AssignStmt Ident "x" Const 0
```
From *identifiers* to *symbols*
Symbol tables

- An *identifier* is a language token type, while the value of an identifier is a *name* (typically a string)

- A *symbol table* maps *names* to *symbols*

- What language constructs create new *names*, and when are those names visible in subsequent parts of the program text?

- What information is useful to track for each symbol?

- Each language will have its own rules about scoping and symbol visibility:
  - Hierarchical
  - Parallel
Scoping & symbol visibility
Major & minor scopes

- Major scopes: reserved for significant constructs in the language
  - Top level program
  - Body of a function/procedure/method
  - Body of a class
  - Module definition
- Minor scope: occur within major scopes
  - Delimited by { and }

- Major scopes typically a unit of resource allocation, while minor scopes can be collapsed together
- Nested scopes can hide access to names from earlier scopes (but they don’t alter the original name/symbol)
var foo integer

foo = 0
var foo = 0
var foo integer
foo = 0
{
    var foo integer
    foo = 1
}
print foo
var foo integer
foo = 0
{
    var foo boolean
    foo = true
}
print foo
Name resolution

- Each scope maintains a list of locally declared identifiers, or names, mapped to symbol table entries

- At each point in the program, certain names are in scope and visible

- Typically search upwards through enclosing scopes to find origin of declaration

- Qualified names allow searching within another contexts

- Importing brings names into scope from other contexts
Qualified names
(hierarchical scoping)

System.out.println("Hello World!")
Qualified names (hierarchical scoping)

```java
System.out.println("Hello World!")
```
import mypkg.Foo;

... Foo f = new Foo();
Importing names

```java
import mypkg.Foo;

... Foo f = new Foo();
```
public, private, protected

class Parent {
    public    int anyone;
    private   int only_me;
    protected int children;
}

class Child extends Parent {
    void m() {
        anyone   = 488;  // allowed
        children = 488;  // allowed
        only_me  =  -1;  // ERROR!
    }
}
Parallel scopes

- Some languages define multiple contexts in which the same name can be used, without conflict
  - C: struct, enum, union
    - Not C++
  - Haskell: data types vs data constructors
struct s {
    int x;
};
struct s s;  // s.s = 488;

Parallel scopes — Haskell

data Atom = Atom;
c :: Atom;
c = Atom;
Symbol table entry

- Original point of instantiation
- Type information:
  - Scalar vs array vs routine
  - Link to record/class/etc. definition
  - For routines: formal parameters and their symbol information
  - For function: return type information
- Visibility modifiers
- Uses (optional)
Implementing a symbol table

- Core operations:
  - Enter into a new scope
  - Exit from a scope
  - Lookup if a name exist in the current immediate scope
  - Lookup if a name exist in the current or a parent scope
  - Put a new name-to-symbol entry in the table
- Multi-level map of names to symbols
  - Stack of hash tables
  - Or, a hash table of lists
Implementing a symbol table

- New levels in the symbol table with typically begin with the start of major and minor scopes
  - Maps naturally between enter & exit and push & pop
- If you perform an Ident-to-Symbol AST transformation, do you still need the hash table anymore?
  - Debuggers depend on knowing what names are visible/accessible at each point in the program