Attributes and binding

- a program might consist of many named items (variables, constants, functions, parameters, labels, etc)
- each such item may have a variety of associated attributes (e.g. data type, stored value, memory address, etc)
- we want to examine what kinds of attribute may be associated with (bound to) an item, when that takes place, and how it may/may not be subsequently altered

Names and named components

- Earlier we discussed the alphabet for a language, the use of regular expressions to describe language tokens
- Names may be divided into keywords, reserved words, predefined names, user defined names
  - Reserved words: have a single, fixed meaning
  - Keywords: have a fixed meaning within a specific context
  - Pre-defined words are words that currently have an associated meaning, e.g. from including a library, but which may be redefined
  - User-defined words are words whose definition is set by the programmer (e.g. for variables, parameters, functions, labels, etc)

Impact of naming rules

- Language naming rules has a significant impact on developers' use of the language
- Are names case sensitive? e.g. are SomeVariable and Somevariable different names or not?
- Is there a maximum length for names? Is there a minimum length? (old basic: can have longer names, but only first three letters count, so MyVar1 and MyVar2 are the same var!)
- What alphabet is available for user-defined names, and does this cause overlap between token types (e.g. if variable names can be alphanumeric can "123" be a variable name?)

Attributes to consider

- Will focus on binding of attributes to variables, but many of these ideas apply when binding attributes to other types of component (functions, identifiers, etc)
  - Name: what word(s) can be used to refer to the item
  - Location: where is the item stored (e.g. a real/abstract memory address)
  - Value: what data value is currently associated with the item
  - Type: what data type is currently associated with the item
  - Lifetime: from time item created to time when it is destroyed
  - Scope: the program segments in which the item is visible/accessible

  (reference environment is inverse of scope: what can you see from line x)

Static and dynamic binding

- When is the value of an attribute determined, and can it change later?
- Static binding: the value of the attribute is fixed/known by compile time, and never changes (e.g. the data type for a C variable, the value of a C constant)
- Dynamic binding: the value of the attribute is determined during execution
  - once the value is determined can it change later or not? E.g. a C variable's value can change after it has been set, but its memory address cannot

Binding examples for variables

Think about the differences between C and lisp: when is a value bound, and can it subsequently be changed?

- Name: static in C, but what about lisp symbol bindings?
- Address/storage location: set when allocated during execution?
- Value: dynamic and changable in both
- Data type: static in C, but changes with the value in lisp
- Scope: think lexical vs dynamic
- Lifetime: is this impacted by lexical vs dynamic scope?

Static vs dynamic typing

- Static typing: the data type is determined by the source code, e.g. an explicitly declared type (int, float, etc) or an implicit type (e.g. in bash everything is a string by default, in perl the type is denoted by a special character in the name)
- Fixed dynamic typing: perhaps the first time we store a value in a variable that determines the data type for the variable, and we cannot subsequently change the type
- Dynamic typing: the variable itself has no specific type, the current “data type” is actually associated with whatever value the variable happens to contain right now

Dynamic typing: pros/cons

- more flexible for developer (can use whatever type is convenient when we need it)
- prevents compile-time checking of data types, so type checking (if done at all) must be done during execution
- can result in uncaught errors (e.g. accidentally storing a string in variable we meant to be an int: no type checking so no warning)
- Some types require more storage space than others: how do we bind storage to a variable? (possibly allocate the stored value in heap, variable is really a pointer to the current heap location)

Variable storage and lifetimes

- Based on the typical use of a run-time stack plus a heap
  - Static variables: created and bound to a location when the program is loaded into memory, at the start of execution, cease to exist when the program ends
  - Stack-dynamic variables: created and bound to a location on the stack at the point where a function call is made, cease to exist when the function ends (stack space deallocated)
  - Heap-dynamic variables: the pointer or reference variable follows standard scope/lifetime rules, but the allocated memory is bound to a location when the dynamic memory request is made, destroyed when the memory free/deallocation request is made

Static (lexical) vs dynamic scope

- Typical lexical scope rules let us see scope of each item directly from the source code (C, Java, etc)
- When resolving names, typically check innermost scope first, gradually expanding scope by scope looking for item with specified name, until eventually reach global scope
- Dynamic scopes (as we've seen in lisp) allow functions to access items declared in their caller's scope (and the scope of their caller, etc etc) in addition to the items normally visible from the lexical scope rules

Dynamic scope pros/cons

- Can eliminate lots of parameter passing, callee can simply access whatever was visible to the caller (or their caller, or their caller, etc up the call chain) ... if they know the name of what to access
- Can create confusing overlap: with lots of function calls and local variable definitions, overlapping names can be confusing/difficult to trace/debug
- Implementing dynamic scoping mechanisms (e.g. writing compilers/debuggers for the language) can be challenging