COSC252: Programming Languages:

Basic Semantics: Environments, Names, Literals, Scope, ...

Jeremy Bolton, PhD

Asst Teaching Professor

GEORGETOWN UNIVERSITY

Outline

I. Review: Some Evaluation Criteria of Languages
    I. - ility's
II. Literals
III. Names
IV. Variables
    I. Bindings
V. Symbol Table
VI. Scope
VII. Environments

Language Evaluation Criteria:

Moving Forward we will use these to evaluate different PL design decisions.

- **Readability**: the ease with which programs can be read and understood
- **Writability**: the ease with which a language can be used to create programs
- **Reliability**: conformance to specifications (i.e., performs to its specifications).
  - Error checking?
- **Flexibility:** Provides many constructs for a wide variety of operations.

Evaluation Criteria: Readability

- Overall simplicity
  - A manageable set of features and constructs
  - Minimal feature multiplicity
  - Minimal operator overloading
- Orthogonality
  - A relatively small set of primitive constructs can be combined in a relatively small number of ways
  - Every possible combination is legal
- Data types
  - Adequate predefined data types
- Syntax considerations
  - Identifier forms: flexible composition
  - Special words and methods of forming compound statements
  - Form and meaning: self-descriptive constructs, meaningful keywords

Evaluation Criteria: Writability

- Simplicity and orthogonality
  - Few constructs, a small number of primitives, a small set of rules for combining them
- Support for abstraction
  - The ability to define and use complex structures or operations in ways that allow details to be ignored
- Expressivity
  - A set of relatively convenient ways of specifying operations
  - Strength and number of operators and predefined functions

Evaluation Criteria: Reliability

- Type checking
  - Testing for type errors
- Exception handling
  - Intercept run-time errors and take corrective measures
- Aliasing
  - Presence of two or more distinct referencing methods for the same memory location
- Readability and writability
  - A language that does not support “natural” ways of expressing an algorithm will require the use of “unnatural” approaches, and hence reduced reliability

Language Design Trade-Offs

- Reliability vs. cost of execution
  - Example: Java demands all references to array elements be checked for proper indexing, which leads to increased execution costs
- Readability vs. writability

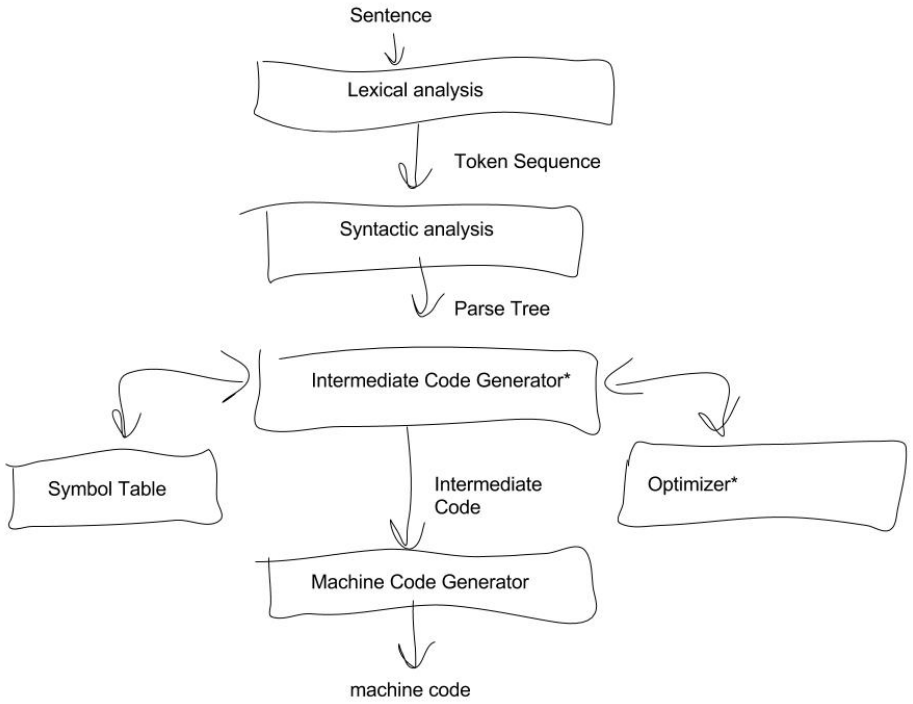
  Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability
- Writability (flexibility) vs. reliability
  - Example: C++ pointers are powerful and very flexible but are unreliable

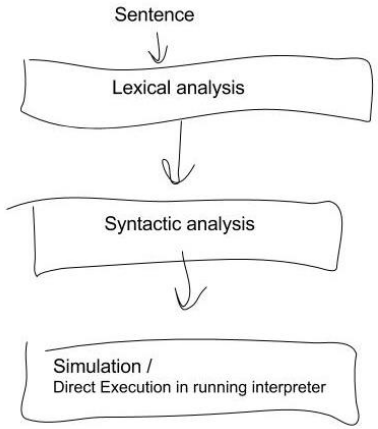
Application of Semantics

- Most programming languages allow programmers to interact with a computer at a highly abstract level.
  - The underlying machine has various low-level components: CPU, memory, …
  - Most programming languages operate at a higher level of the computing abstraction.
    - For example: the abstraction for a memory location is a variable.
    - Operating at this level of the abstraction conceals low level details from the programmer which eases the programming burden.
  - However, concept of semantics, is intrinsically defined in terms of the state of the machine.
    - Yes – the goal of a programming language is to communicate with a machine – give precise instruction.
      - Intuitively, the instruction must therefore be in
        1. In a format understandable to the machine
        2. Performable by the machine

Semantics at different levels of the computer abstraction

- Application of Semantics: Compiler vs. Interpreter
  - Compiler
    - A compiler often translates source code to machine code. Thus communicating at the lowest level of the abstraction.
    - Thus the application of semantics is *the creation of machine code* (to be performed by the computer)
  - Interpreter
    - The interpreter is a running program that "simulates" the execution of the source code. It interprets the commands in the source code and then directly performs those commands (or commands similar to those which are available).
    - Thus application of semantics is *the execution of interpreted commands.*
- *Examples to come …*





Semantics of tokens

- Application of semantics is often referred to as *evaluation*.

- Example literal:
  - What is the semantics of “1”?
  - Conceptually the string “1” refers to the quantity 1.
  - Semantic application (evaluation) of “1” in a computer, refers to the binary representation of “1” in a computer.
    - Load Register with 0000 0000 …. 0001

- Example:
  - What is ‘c’?
  - Conceptually: the character c
  - Semantic application (evaluation) in a computer, refers to the binary representation of ‘c’

Semantics of variables

- Semantics of most literals is relatively simple.
- Semantics of variables requires some extra overhead to maintain our computing abstraction.
- What is a variable?
  - What attributes are needed to fully characterize or uniquely define each variable in a program?

Variable Attributes

- Name
  - A means to refer to the variable at a high level of the abstraction.
    - Programmer does not need to know the variables actual location, number of bits, …
- Address
  - The actual location of the variable in memory
  - L-value
- Value
  - The value or binary sequence at variables address
  - R-value

X = 5 + X; // l-value vs. r-value

Variable Attributes

- Types
  - Are a mapping from a set of binary sequences to a value
    - Helps define interpretation / evaluation of binary sequence
    - EG , $int: \{0,1\}^{32} \rightarrow \left[-2^{31}, 2^{31}-1\right] \cap \mathbf{Z}$
  - Generally implies the “size” of a variable (how many bits)

Variable Attributes

- Lifetime
  - Variables generally do not live forever.
  - They may be explicitly or implicitly allocated and deallocated given the semantics of a programming language
    - The time between allocation and deallocation is generally referred to as the lifetime of a variable.
- Scope
  - Refers to the accessibility of a variable.

Binding

- A variable is *bound* to an attribute at *binding time*
  - Static binding: binding occurs before runtime
  - Dynamic binding: binding occurs during runtime

- C++ Example:
  - int i = 0;

  - type of i is bound statically (at compile time).
  - address may be statically determined (load time) or dynamically determined (at runtime) if local variable
  - lifetime and scope bindings will also depend on whether the variable is global or local.

Type Binding

- A variable must be bound to a data type before it can be referenced (and evaluated)
  - Type determines size and maps to value
  - Type binding can be static or dynamic

- Static type binding
  - Explicit declaration
    - int i = 0; // eg c++
  - Implicit declaration. Type is never explicitly declared, but is determined before runtime. But How?
    - i = 0;

Implicit Static Type Binding

Example: i = 0;

– How can type be determined?

- Type is bound by compiler / interpreter which tracks types, e.g., using an attribute grammar.
  – Also known as *type inference.* Note*:* Variable must be initialized generally.
- To simplify, some PLs require specific naming formats for particular types.
  – E.G. Perl variables beginning with a $ are scalars (string or numeric)
  $var = ‘Hello’
- Depending on the language, initialization may be required to determine type.

Dynamic Type Binding

- Dynamic type binding
  - Variables type is not explicitly noted.
  - Type is determined at runtime

  - **Example** (python):

```
def f(x):
    z = x
    return z
```

  - When an assignment statement is evaluated. The right hand side (RHS) will have an associated type. This type is generally bound to LHS variable when the assignment occurs *during runtime.*
  - Observe.
    - A variables type may change during the course of the running program.
    - Memory allocation concerns … we do not know the size of x or z at compile time.

Dynamic vs. Static Type binding

- Tradeoff
  - Dynamic
    - Increased flexibility and writeability for programmer
    - The ability to error check is reduced and thus reliability is reduced.
    - Less efficient execution
      - Type checking and conversions must be done at runtime – slow!
      - Memory allocation issues … more to come!
  - Static
    - Increased readability, decreased flexibility
    - More efficient execution
    - Simple memory allocation – size is known at compile time!

Lifetime

- The time a variable is bound to its address (time between allocation and deallocation) is its lifetime.

- 4 categories
  - Static variables
  - Stack-dynamic variables
  - Explicit-heap-dynamic variables
  - Implicit-heap-dynamic variables

Static Variables (lifetime)

- Static Variables
  - Variables to be used throughout program execution
    - E.G. global variables
  - Static variables are bound before runtime (during load time), and remain bound until the program execution terminates.
  - Tradeoff
    - Efficient
      - Addressing a static variable is *direct* (address is constant)
      - No runtime overhead for allocation and deallocation
    - Reduced flexibility – no recursion with static variables
    - Reduced readability – use of non-local variables can reduce readability of code.

Stack-Dynamic Variables (lifetime)

- Variables allocated on the runtime stack
  - Type is generally bound statically: the storage space needed for local variables is determined statically. As a result, size is fixed.
  - Address is bound dynamically (on the *runtime* stack).
  - Tradeoff
    - Permits recursion
    - Increases readability of code
    - There is time overhead for allocation and deallocation on the stack. Indirect addressing is required which may result in small delay. Delays noted are minimal.

Explicit Heap-Dynamic Variables (lifetime)

- Variables that are explicitly allocated on the Heap.
  - Allocation: new // c++
  - Deallocation: delete // c++
- The heap is relatively disorganized (compared to stack), and the management of this memory is often left to the programmer; whereas the stack is self managing.
- Tradeoff
  - Allows for constructs *whose sizes may change dynamically*: e.g. linked lists; and constructs whose *sizes are not known at compile time*
  - Con: memory management is left to the programmer.
  - Given the disorganized state of the heap, overall memory management and overhead for allocation and deallocation is slower as compared to runtime stack.

Implicit Heap-Dynamic Variables (lifetime)

- Not explicitly assigned to heap by programmer, but assigned to the heap nonetheless. Determined by programming language.
  - As result all memory management must is generally handled by the runtime environment (slow)
  - E.G.
    - All objects in Java are implicit heap dynamic variables.
    - Lists in Python are implicit heap dynamic variables.
- Tradeoff
  - Eases burden of programming: improves flexibility and writeability.
  - Reduced error checking reduces reliability.
  - Overhead is slow – run time environment must manage garbage collection (when to deallocate variables no longer being used).

Scope

- Scope refers to the accessibility or visibility of a variable or name.
- Scoping rules for languages vary.
  - Static scope – aka lexical scope.
    - Scope is determined statically, at compile time.
    - It is generally determined given the lexical structure of the code.
    - If scopes are embedded, accessibility of variables is facilitated by tracing back to parent scopes (the static parent) or back to static ancestors.
      - A parent scope of a child scope is the scope where the child scope is lexically declared.
  - Dynamic scope – zany !
    - We will revisit during discussion of the runtime stack

Static (lexical) Scope

- Global scope is parent scope for all scopes in C (the root scope).
- “{ }” are the scoping operators.

```
#include <iostream>
using namespace std;
// global scope
const double pi = 3.14;

int   f( int input){
// ‘f’ scope – all variables in global and declared in ‘f’ scope are accessible here
int val = 3;
while( input < 0){  // local ‘while’ scope: all* variables declared in ‘while’, ‘f’ and global scopes are accessible here
        int sum = 1+input;
        input --; val+=sum;}
return pi *input;}

int main(){ // ‘main’ scope– all variables in global and main scopes are accessible here
        int val = 2;
        cout << f(5)*val<<endl;
        Return 0;}
```

GEORGETOWN UNIVERSITY

Static Scope

```
#include <iostream>
using namespace std;
// global scope
const double pi = 3.14;

int   f( int input ){
// 'f' scope – all* variables in global and 'f' scope are accessible here
double pi= 3;
while( input ){  // local 'while' scope all* variables defined in 'while', 'f' and global scopes are accessible here
        input --; pi++}
        return pi;}

int main(){ // 'main' scope– all variables in global and main scopes are accessible here
        int val = 2;
        cout << f(5)*val<<endl;
        Return 0;}
```

When executing function f, global pi is still alive, but is not accessible (in C++). When pi is referenced, the local symbol table is first searched, and pi is found. In a sense, the local declaration of pi has blocked access to the global pi variable (which would otherwise be accessible.).

Some programming languages allow for scope labeling, to provide a solution to this issue.

Accessibility and Scope (sequential execution)

- Accessing a variable before (in sequential execution) it has been defined is generally not permitted.
  - Some PLs require that all variables be declared at the beginning of a new scope (code block).
    - C ’89
  - Some PLs allow variables to be defined anywhere a statement can appear.
    - Variable cannot be accessed before defined.
      - C ’99
    - Variable can be accessed before (sequentially) it is defined, as long as defined in same scope / block
      - javascript

The symbol table

- The symbol table contains pertinent information related to all (symbols) names: variables, functions, …
- For a variable, this may include all info related to the 6 attributes
  - name, address, type, value, lifetime, scope
- Lexical scope can be maintained using a stack-like structure or a tree-like structure

Example (crude) Symbol Table

Symbol Name	Variable info	Variable info	Variable info
val	(5, int, 0x00a0b0, f-scope)	(, double, 0x0cc0b0, q-scope)	
iter	(NULL, Node*, 0x00acb0, f-scope)		
pi	(3.0, double, 0x00acb0, f-scope)	(3.14, double, 0x00acb0, global-scope)	

- Here each column represents 1 variable.
- Many variables may share the same name and thus each row may have multiple entries
- This design of a symbol table is crude, but helps to illustrate the goals of the table.
- Lets look at some better design schemes …

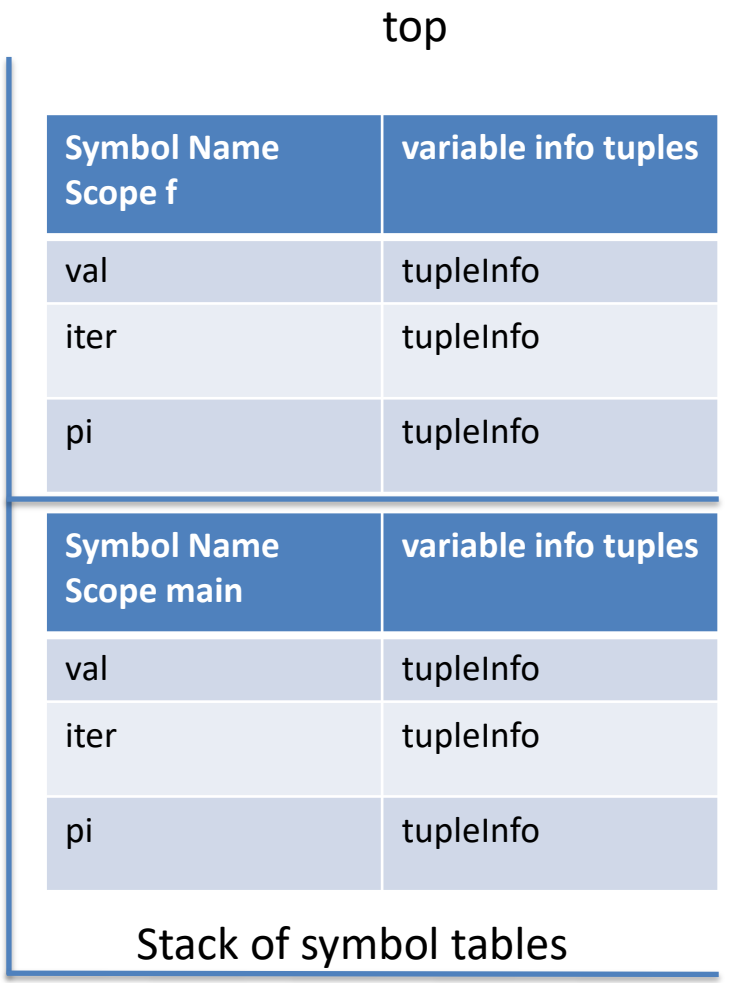
Symbol Table

- Scope implemented as a stack
  - One option: One symbol table, where each entry in the symbol table is a stack.
    - Every time a new function is called or a new scope is entered, perform a push for each new variable encountered, every time you exit a scope or return from a function call, perform a pop.
      - Dynamic Scope! Depends on the flow of execution.
      - Finding a variable (by name) – need only check the top of the stack for the entry corresponding to name.
    - Similarly one might implement a stack of symbol tables

Symbol Name	Stacks of variable info tuples
val	Stack Head
iter	Stack Head
pi	Stack Head

Symbol Table

- Stack
  - Another Option: Scope is implemented as tree of symbol tables (next slide).
    - Static Scope: A parent symbol table is a static parent (with respect to static or lexical scope).
    - Dynamic Scope: A parent symbol table is the “calling scope” (with respect to the flow of execution, runtime stack).
    - Finding a variable
      - Search current symbol table – symbol table associated with current scope
      - If found great, if not, iteratively check parent scopes until name is found.

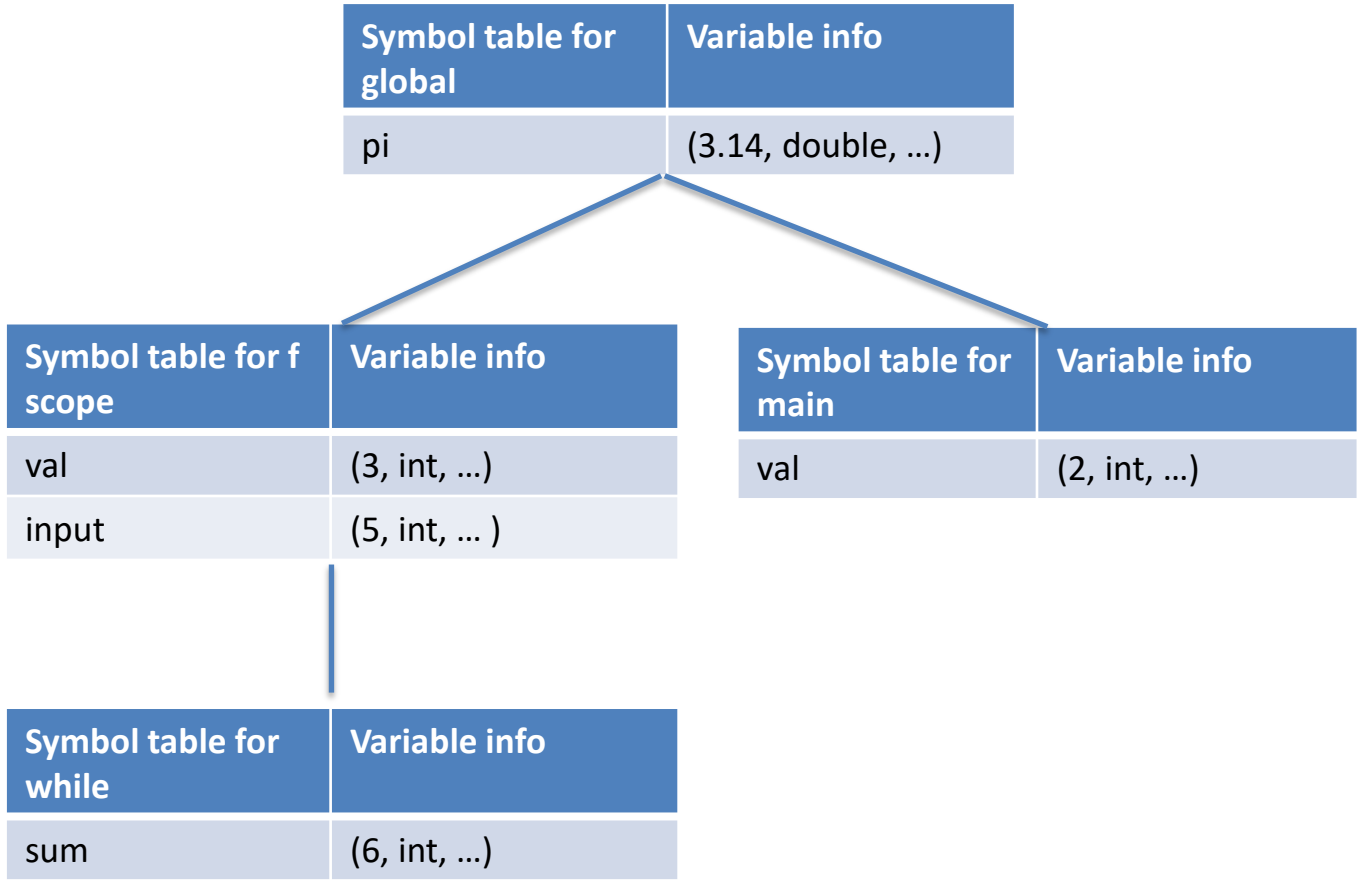


Tree of Symbol Tables: Example

```
#include <iostream>
using namespace std;
// global scope
const double pi = 3.14;

int   f( int input){
// 'f' scope – all variables in global and 'f' scope are accessible here
int val = 3;
while( input > 4){  // local 'while' scope all variables defined in 'while', 'f' and global scopes are accessible here
        int sum = 1+input;
        input --; val+=sum;}
return pi *input;}

int main(){ // 'main' scope– all variables in global and main scopes are accessible here
        int val = 2;
        cout << f(5)*val<<endl;
        Return 0;}
```



GEORGETOWN UNIVERSITY

Referencing Environment

- The referencing environment for a statement is the collection of all names that are visible (accessible).
  - For a statically scoped language. The referencing environment of a statement includes the variables declared in its local scope plus those declared in ancestor scopes.

Appendix

GEORGETOWN UNIVERSITY