Bali Types in your compiler

A how to

Things to consider

- Categories of Types:
  - Type “Signatures” (or Archetypes)
    - Primitive types
      - Integer
      - Boolean
      - Character
    - Function types
    - Class Types
      - (There are NOT in part 3)
  - Pointer types

A bali example:

```
class Entry
{
    int main()
    {
        my int a1;
        my int* a2;
    }
    char c(int a, boolean* b)
    {
    }
}
```

The symbol tables

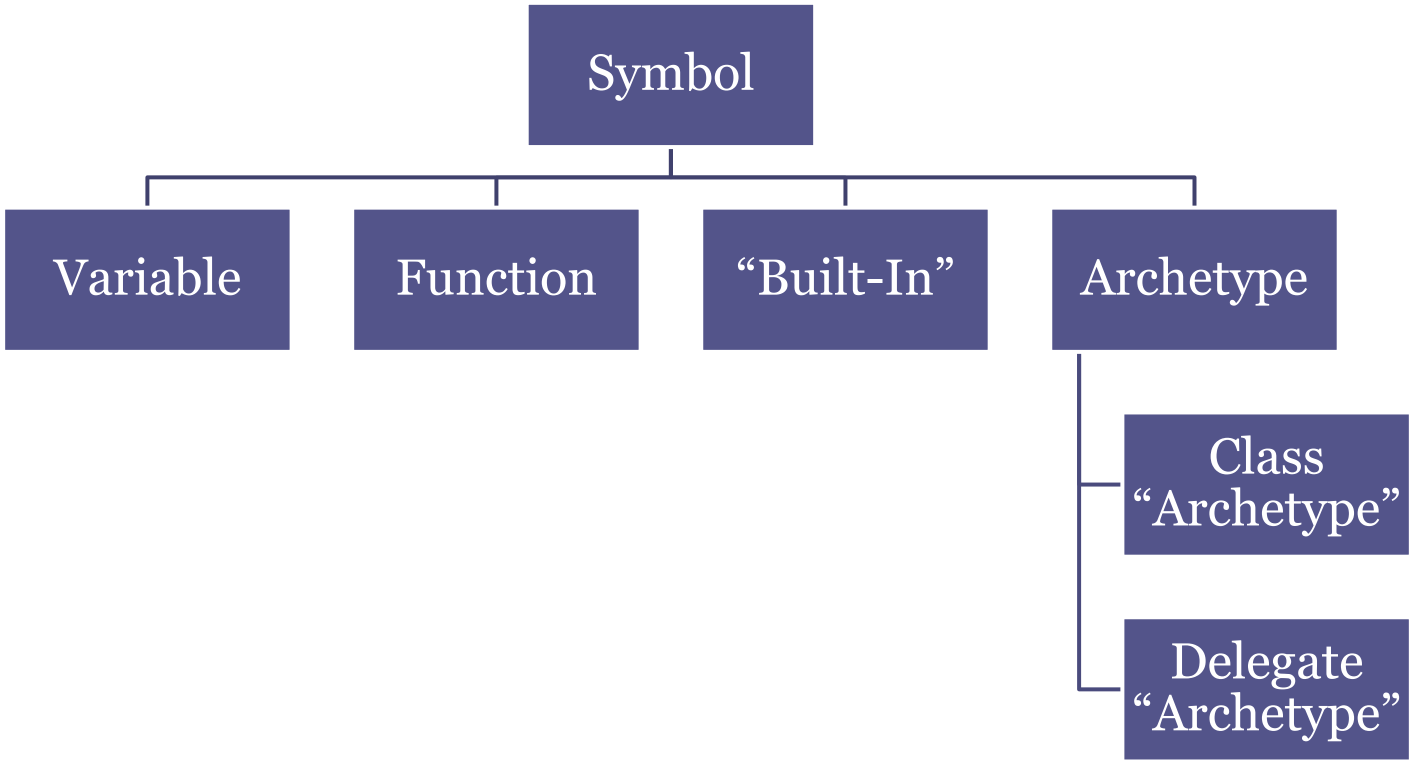
- Not considering class symbol tables (part 3)
- Global Symbol Table:
  - Int type, Boolean type, Char type
  - ReadInt, ReadChar, ReadString
  - “Functions”
    - c
    - main
- Symbol Table for “main”
  - a1 -> (int, SO = 2)
  - a2 -> (int*, SO = 3)
- Symbol Table for “c”
  - a -> (int, SO -2)
  - b -> (boolean*, SO= -1)

Symbol tables ARE mappings!

- What are we mapping from?
  - Strings
- What are we mapping to?
  - Symbols!

- What can we map?
  - Archetypes (eg: int, boolean, *CustomClass*)
    BUT NOT: pointers!!
  - Variables
  - Functions
  - "Built-In-Special" variables (eg: ReadChar)

Creating a type hierarchy



Suggestion: use Enums to differentiate known compile time differences?
Also: Where do pointers fit in? Create a new class (eg: Type) that references an “Archetype” and has a field to denote the “pointer level”.

What is a Delegate Archetype?

- Primitive types can be represented as Simple Archetypes (ie: by setting the type to some enum)
    - There is no other information that we need to know about primitive types
- For Functions/Delegates?
    - Return Type
    - Argument Types
        - Number of arguments
- For Classes
    - Field Members
    - Methods

Symbol Tables for functions

- Functions have their own symbol tables
- But the first block can not have variable names that conflict with function parameters
- Function and first block "share" a symbol table?
- Solutions:
  - All functions have a SINGLE initial block
  - Have the function put the arguments into that initial block's symbol-table

Different "strokes"(symbol tables) for different "folks" (AST nodes)

- Different types of AST Nodes do different things
  - eg: They print out different SaM code
- Different Symbol tables need to do different things
  - eg: For Blocks, the Symbol Table might "automatically" number the stack offset for variables.
  - You might need to use a different offset when doing classes
  - Or no numbering at all for the global symbol table?
- Solutions:
  - Can create different symbol tables that do different things when added to.
  - Do you ever have to remove from a symbol table?
    - NO!!
  - Also: Parameters are numbered differently in the symbol table than variables
  - Only certain types of symbols can go in certain symbol tables (eg: "Built-Ins" can't go in a Block's Symbol Table)

Working with nested scopes

- AST Nodes that have scope (or create a dedicated symbol table)
  - Global (eg: Program)
  - Class (Not for this part)
  - Method (though might use the block's symbol table)
  - Block
- Since the Method uses the block's symbol table, we need only consider blocks
- Solutions:
  - Allocate variables "on demand"
  - If no return statement, just de-allocate variables at end of block (ADDSP)

- Solutions:
  - Can't conditionally allocate variables in a Block's scope, except via "Return Statement"
  - For Return (in nested block)
    - Keep track of variable allocations at the "method" level
    - Before jumping to "end of function", ADDSP the number of variables allocated at the method level
    - At the end of each block, subtract variable allocation of block from the method level
    - DON'T include parameters

AMS Demo and Questions