Symbols and Names

Syntax Directed Translation construction of an AST during a parse is insufficient for modern languages.

One example of where SDT is insufficent is in symbol and identifier management. In a bottum-up parse, you can't perform robust checks on variable use until you've seen the variable declaration and know the full structured scope of the program. You need the full tree for this.

Symbols and Names

static scoping Scope of an identifier does not change during the compilation process.

block structure Many languages use specific syntax to create and destroy nested scopes (bracing, function definitions, control structures, indentation)

The static scope of an identifier includes its defining block (after declaration) and any nested scopes within the block.

current scope is the innermost nested scope

active scopes (open scopes) the current scope and its containing (outer scopes)

Transient Symbol Tables

Information used for each symbol (identifier) encountered:

1. Name (sort of)
2. Type
3. Attributes (constant?, size?, storage requirements?)
4. Scope (in some symbol table implementations)

As identifiers are verified and their use checked, (some) information about the identifier is copied from the symbol table to global data structures or **relevant AST nodes** for future processing.

After a scope block has been validated, it is typically destroyed and not rebuilt for the balance of the compilation process (because we've stored the information we must preserve. . .).

Names

Instead of explicit names, the symbol table stores a reference to the name in a global namespace — perhaps an expandable buffer of characters.

offset	0	1	2	3	4	5	6	7	8	9	10	11	...
char	g	v	_	o	v	e	l	o	c	i	t	y	...

1. References are simple tuples: $(offset,\ length)$.
2. Names go in and **don't come out** (interned until compilation is finished)
   (why? we'll need these names when labeling dynamic library entry points, memory, exported data, ...)
3. Depending on the implementation, the same character region in the namespace can be shared amoung different identifiers.
   `veleocity` = $(4, 8)$, `i` = $(9, 1)$, `g` = $(0, 1)$, `city` = $(8, 4)$.
4. Two different references can hold the same name $(1, 1)$ and $(4, 1)$ are both little `v`.

Symbol Table Interface

`openScope(), closeScope()` as one might expect — accumulated scope information is thrown away when closed.

`enterSymbol(name, type)` when declarations are discovered — fails when a symbol of the same name pre-exists in the same scope.

`retrieveSymbol(name)` used when validating variable (identifier) use. Has it been declared? Initialized? Are we storing a character string into a `boolean` type?

`declaredLocally(name)` the identifier has been declared in the innermost scope.

Symbol Table Implementation

One global table or many symbol tables in a stack?

Downside to a stack of tables: most lookups are found in the innermost (current) scope and the global scope. Having to search all symbol tables between the two scopes can be inefficient.

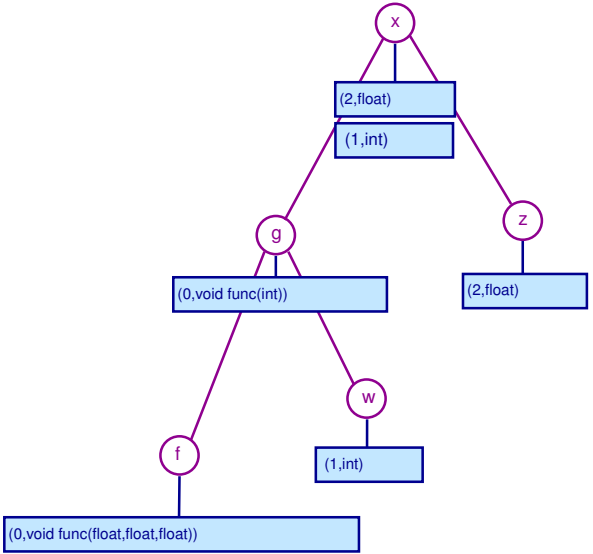
Symbol Table Implementation

One global table or many symbol tables in a stack?

General design of one type of symbol table implmentation:

- A $(key, value)$ data structure $SymbolTable$;
  key = string name (BEWARE using namespace refs — because two different refs, (eg: (1,1) and (4,1) can represent the same name value, `v`) — your key comparison logic must do more than compare offset and length in the namespace.)
  $value$ = **stack** of symbol meta data structure $(scope, \{type, attributes, \ldots\})$.
- `openScope()`: increment table-wide current scope value (0, 1, 2, 3, ...)
  0 is global scope, higher values are increasingly nested scopes, highest value is current scope
- `enterSymbol(name,metaData)`: create $SymbolTable[name]$ as an empty stack if DNE, push $(scope, \{metaData\})$ onto stack.
- `declaredLocally(name)`: if $SymbolTable[name]$ exists, is it's stack top the current scope?

One Tree, Multiple Scopes




```
import f(float,float,float)
import g(int)
{
     int w, x
     {
          float x, z
          f(x, w, z)
     }
     g( x )
}
```

Type Checking

Type checking is one of the obvious reasons to maintain a scope aware **symbol table**. Type checking in compilers can be complicated, but it won't be in this course — we'll rely on your experience with C++, Java, Python to *scaffold expectations* (and wth does that mean?).

The **ZOBOS** project (soon!) will have you implement several type checking tests on a semi-conventional programming language. Type checking is very language definition dependent, but here are some of the basic tests that fall under the category:

1. What operations (arithmetic or otherwise) are permitted between variables or constants of various types.
2. What types will the compiler automatically coerce (cast) into other language types?
3. What types may the programmer explicitly coerce?
4. What types may be compared to each other for value equality? (Is `0==0.0`?)
5. Is the identifier to the left of an assignment a valid storage location for the RHS value of =? (IOW: When is `counter = 3` allowed?)

Section § 8.8 of the text is a not-too-deep plunge into such questions, not required reading, but you may find it interesting in your "spare time."

Deep Dive

Along with the reading, you should be ready for `lga-symbol-tables.pdf` .

Next, we take a deep dive into a **very efficient symbol table implementation** and how it can be closely mimicked in our modern high level languages (HLLs).

`show_efficient_symtable.pdf`