Today

- Symbol table implementation strategies
  - For a given scope (collection of name/value pairs)
  - For nested scopes
- Managing types
  - Taxonomy
  - Representation
  - Terminology
  - …

Symbol table implementation strategies

- Abstractly, it’s simple: mapping from names to the information
- Concretely, there are lots of choices, each with different performance consequences
- So, we’ll take a brief trip down the CSE326 lane…

Three basic options

A. Linked list (or dynamic array) of key/value pairs
B. Sorted binary search tree
C. Hash table

Complexity of each option

<table>
<thead>
<tr>
<th></th>
<th>Enter</th>
<th>Lookup</th>
<th>Space cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Linked lists</td>
<td></td>
<td>$O(1)$</td>
<td></td>
</tr>
<tr>
<td>B. Binary search tree</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C. Hash table</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A few more issues

A. Linked lists must have keys that can be compared for equality
B. Binary search trees must have keys that can be ordered
C. Hash tables must have keys that can be hashed (well)
Summary

- In general
  - Use hash tables for big mappings
  - Use binary tree or linked list for small mappings
    - Jon Bentley’s story about a BASIC interpreter
  - Ideally, use a self-reorganizing data structure

Implementing nested scoping

- Each scope (instance of `SymTabScope`) keeps a pointer to its enclosing `SymTabScope` (_parent)
- In addition, each scope maintains “down links”, too (_children)

Types

- Types are abstractions of values that share common properties
  - What operations can be performed on them
  - (Usually) how they are represented in memory
- Type checking uses types to compute whether operations on values will be legal
  - That is, will the operations compute a type of value that is legal in that context
- Types usually guide how compilation proceeds

Taxonomy of types

- Basic/atomic types
  - `int`, `bool`, `char`, `real`, `string`, ...
  - `enum(v1, v2, ..., vn)`
  - User-defined types: `Stack`, `SymTabScope`, ...
- Type constructors
- Parameterized types
- Type synonyms

Type constructors

- `ptr(type)`
- `array(index-range, element-type)`
- `record(name1:type1, ..., nameN:typen)`
- `tuple(type1, ..., typen)` or `type1 × ... × typen`
- `union(type1, ..., typen)` or `type1 + ... + typen`
- `function(arg-types, result-type)` or `type1 × ... × typen → result-type`

Parameterized types:

- `Array<T>`
- `HashTable<Key,Value>`
- ...

Type synonyms:
Give alternative name to existing type
- typedef SymTabScope* SymTabReg

Type checking terminology
- Static vs. dynamic typing
  - Static: checking done prior to execution (e.g., compile-time)
  - Dynamic: checking during execution
- Strong vs. weak typing
  - Strong: guarantees no illegal operations performed
  - Weak: no such guarantee
- Caveats
  - Hybrids are common
  - Mistaken usages of these terms is common
  - Ex: "untyped", "typeless" could mean "dynamic" or "weak"

Fill in with real languages

<table>
<thead>
<tr>
<th>Static typing</th>
<th>Dynamic typing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong typing</td>
<td>Weak typing</td>
</tr>
</tbody>
</table>

Type checking
- Assume we have an AST for the source program
  - It is syntactically correct
  - The symbol table has been computed
  - Now we need to check to see if it meets the type constraints of the language
  - Ex: `a := 3 * b + fork(c + 3.14159)`
  - What is the type of `a`, `b`, and `c`? What type does `fork` return? What type does `fork` accept? What happens when `c` is added to a `float`? What happens when `c` is multiplied by `3`? What happens when `fork`'s result is added to `3 * b`?

Strategy
- Traverse AST recursively, starting at root node
  - Most work is on the bottom-up pass
  - At each node
    - Recursively typecheck any subnodes
    - Check legality of current node, given the types of the subnodes
    - Compute and return result type of current node, if any

Example:
3 * b + fork(c + 3.14159)
Top-down information also:

From enclosing context

- Need to know types of variables referenced
- Must pass down symbol table during traversal
- Legality of (for instance) break and return statements depends on context
- Must pass down whether in loop, what the result type of the function must be, etc.

Representing types in PL/0

```cpp
class Type {
    virtual bool same(Type* t);
};

class IntegerType : public Type {
};

class BooleanType : public Type {
};

class ProcedureType : public Type {
    TypeArray* _formalTypes;
};

IntegerType* integerType; BooleanType* booleanType; ...
```

Type checking in PL/0: overview

```cpp
Type* Expr::typecheck(SymTabScope* s);
void Stmt::typecheck(SymTabScope* s);
void Decl::typecheck(SymTabScope* s);
Type* LValue::typecheck_lvalue(SymTabScope* s);
int Expr::resolve_constant(SymTabScope* s);
Type* TypeAST::typecheck(SymTabScope* s);
```

Today++

- A detailed look at how PL/0 actually implements some of these typecheck methods