CSE401: Semantic Analysis (B)

David Notkin
Autumn 2000

UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 2

Today

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

UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 3

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…

UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 4

Three basic options

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

UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 5

Complexity of each option

	Enter	Lookup	Space cost
A. Linked lists	O(1)		
B. Binary search tree			
C. Hash table			

UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 6

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)

UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 7

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

UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 8

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`)

UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 9

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

UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 10

Taxonomy of types

- Basic/atomic types
  - `int, bool, char, real, string, …`
  - `enum`(v_1, v_2, …, v_n)
  - User-defined types: `Stack, SymTabScope, …`
- Type constructors
- Parameterized types
- Type synonyms

UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 11

Type constructors

- `ptr`(type)
- `array`(index-range, element-type)
- `record`(name_1:type_1, … name_n:typen)
- `tuple`(type_1, …, type_n) or $\text{type}_1 \times \ldots \times \text{type}_n$
- `union`(type_1, …, type_n) or $\text{type}_1 + \ldots + \text{type}_n$
- `function`(arg-types, result-type) or $\text{type}_1 \times \ldots \times \text{type}_n \rightarrow$ result-type

UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 12

Parameterized types:

Functions returning types

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

UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 13

Type synonyms:

Give alternative name to existing type

- `typedef SymTabScope* SymTabReg`

UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 14

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"

UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 15

Fill in with real languages

	Statically typed	Dynamically typed
Strong typing		
Weak typing		

UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 16

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 `b` is multiplied by `3`? What happens when `fork`'s result is added to `3 * b`?

UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 17

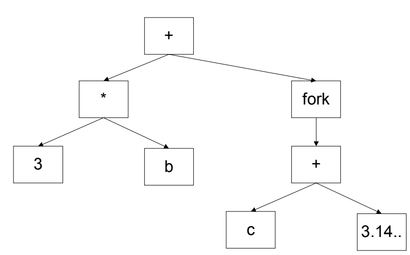
Strategy

- 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

UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 18

Example:

`3 * b + fork(c + 3.14159)`



UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 19

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.

UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 20

Representing types in PL/0

```
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; …
```

UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 21

Type checking in PL/0: overview

```
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);
```

UW CSE401 AQ 2000 • D. Notkin • All rights reserved • Semantics B • Slide 22

Today++

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