CSE401: Semantic Analysis

Larry Snyder
Spring 2003

Semantic analysis

- Perform final legality checking of input program
  - Properties not checked by lexical or syntactic checking
    - Ex: type checking, ensuring break statement is in a loop, etc.
  - "Understand" program well enough to do the back-end synthesis activities
    - Ex: relate particular names to particular declarations

Symbol tables

- Key data structure (at compile time, not run time)
  - Produced (and used) during semantic analysis
  - Used during code generation
  - Stores information about names used in the program
    - Declarations add entries to the symbol table
    - Uses of names look up appropriate symbol table entry

What information about names?

- Kind of declaration
  - var, const, proc, etc.
- Type
- For const: keep value
- For var: Where allocated in memory?
  - Static, stack, heap? Offset?
  - Not computed initially, but later on
- For formal parameter: passed by-value, by-ref...

Prototype compiler structure

Example: a PL/0 DeclList

```plaintext
var x : int;
var q : array[20] of bool;
procedure foo(a : int); begin .. end foo;
const z : int = 10;
```
PL/0 symbol table entries

class SymTabEntry {
    public:
    char* name();
    Type* type();
    virtual bool isConstant();
    virtual bool isVariable();
    virtual bool isFormal();
    virtual bool isProcedure();
    virtual int value();  // const only
    virtual int offset(SymTabScope* s); // var only
}

SymTab subclasses

class VarSTE : public SymTabEntry { ... }
class FormalSTE : public VarSTE { ... }
class ConstSTE : public SymTabEntry { ... }
class ProcSTE : public SymTabEntry { ... }

Nested scopes: Example

procedure foo(x:int, w:int);
var z:bool;
const y:bool = true;
procedure bar(x:array[5] of bool);
var y:int;
begin
    x[y] := s;
end bar;
begin
    while s do
        var z:int, y:int;
        y := x * s;
        end;
        output := x * y;
end foo;

Nested scopes: How to handle?

- What happens when the same name is declared in different scopes?
- This is first a question of language design: what is the defined semantics?
  Two standard choices
  - Lexical (static) scoping: use the block structure of the program
  - Do you remember choice #2 from 341?

Nested Scopes: Lexical/static

- The syntactic (block) structure of the program determines how names are resolved
- Given a name in a block
  - The nearest enclosing block with a declaration for that name is the relevant declaration
  - If none, it's an error

Nested scopes: Dynamic

(c) Larry Snyder & UW CSE, 2003
Lexical scope and symbol tables
- Each scope has its own symbol table
- Logically, for a block-structured program, there is a tree of symbol tables
  - Root = outermost block

Tree of symbol tables

Lexical scope and symbol tables
- Each scope has its own symbol table
- Logically, for a block-structured program, there is a tree of symbol tables
  - Root = outermost block
- But at a given point in the program, only part of the tree is relevant
  - Current block == X
  - Nearest enclosing block == parent(X)
  - Next nearest == parent(parent(X))
  - Etc., up to root

Nested scope operations
- When encounter a new scope during semantic analysis
  - Create a new, empty scope
  - Its parent is the current scope (that of enclosing block)
- When encounter a declaration
  - Add entry to the current scope
  - Check for duplicates in the current scope only (why?)
- When encounter a use
  - Search scopes for declaration: current, its parent, grandparent...
- When exiting a scope
  - Parent becomes current again

PL/0 symbol table interface
class SymTabScope {
  public:
    SymTabScope(SymTabScope* enclosingScope);
    void enter(SymTabEntry* newSymbol);
    SymTabEntry* lookup(char* name);
    SymTabEntry* lookup(char* name,
                           SymTabScope*& retScope);
}
Symbol tables: Implementation
- Abstractly, it's simple:
  a mapping from names to information, aka
  key/value pairs
- Concretely, there are lots of choices, each
  with different performance consequences,
  e.g.
  - Linked list (or dynamic array)
  - Binary search tree
  - Hash table
- So, we'll take a brief trip down CSE326
  memory lane...

Symbol tables: Complexity

<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></td>
<td>O(1)</td>
</tr>
<tr>
<td>B. Binary</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>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>

Symbol tables: Other issues
- Linked lists must have keys that can
  be compared for equality
- Binary search trees must have keys
  that can be ordered
- Hash tables must have keys that can
  be hashed (well)
- Hash table size?

ST: Implementation Summary
- In general
  - Use a hash table for big mappings
  - Use a binary tree or linked list for small
    mappings
  - Ideally, use a self-reorganizing data
    structure

Types
- Types are abstractions of values that share
  common properties
  - What operations can be performed on them
  - (Usually) how they are represented in memory
- Types usually guide how compilation
  proceeds

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

(c) Larry Snyder & UW CSE, 2003
Type constructors
- ptr(type)
- array(index-range, element-type)
- record(name1:type1, ..., nameN:typeN)
- tuple(type1, ..., typeN) or type1 x ... x typeN
- union(type1, ..., typeN) or type1 + ... + typeN
- function(arg-types, result-type) or type1 x ... x typeN → result-type

Parameterized types
Functions returning types
- Array<T>
- Stack<T>
- HashTable<Key,Value>
- ...

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

Type checking
- A key part of language implementation
  - Semantic analysis phase, linking, and/or runtime
  - Verifies that operations on values will be legal
    - I.e., they compute values that will be legal in context
- Examples
  3 + 4  
  3 + x  
  3[x]  
  3 + TRUE 

Type checking terminology
- Static vs. dynamic typing
  - Static: checked prior to execution (e.g., compile-time)
  - Dynamic: checked 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

(c) Larry Snyder & UW CSE, 2003
**Type checking strategy**

- Traverse AST recursively, starting at root node
- Most work is on the bottom-up pass
- At each node
  - Recursively type check any subtrees
  - Check legality of current node, given children's types
  - Compute and return result type (if any) of current node

**Top-down information also:**

From enclosing context

- Need to know types of variables referenced
- Must pass down symbol table during traversal
- Legality of (e.g.) break and return statements depends on context: 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; // predefined instance
BooleanType* booleanType;
```

**PL/0 Type Checking**

```cpp
procedure foo(x:int, w:int); var : bool; const y: boolean = true;
procedure bar(x:array[1] of bool); var y:int;
begin
    x[y] := y;
end bar;
begin
    while z do
        var x:int, y:int;
        z := x * y;
        output := x + y;
        end foo;
end foo;
```

**Ex: 3 * b + fork(c + 3.14159)**

- Syntax:
  - `D: int`
  - `C: float`
  - `fork: float -> int`
- Node:
  - `+`
  - `3`
  - `b`
  - `fork`
  - `c`
  - `3.14`

**PL/0 type checking: 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);
```

(c) Larry Snyder & UW CSE, 2003
Type checking PL/0 expressions

A simple case: integer literals (like "0" or "-17")

```c
Type* IntegerLiteral::typecheck(SymTabScope* s) { 
  return integerType; 
}
```

Type checking operators

```c
Type* BinOp::typecheck(SymTabScope* s) { 
  Type* left = _left->typecheck(s); 
  Type* right = _right->typecheck(s); 
  switch(_op) { 
    case PLUS:case MINUS:case MUL:case DIVIDE: 
      if (left->different(integerType) || 
          right->different(integerType)) { 
        Plzero->typeError("args not int type"); 
      } 
      break; 
    case EQUALS:case NEQ:case LESS:case LESS_EQUAL: 
      if (left->different(right)) { 
        Plzero->typeError("args not same type"); 
      } 
      break; 
    default: 
      Plzero->fatal("unexpected BINOP"); 
      return NULL; // not actually executed 
    }
}
```

Type checking assignments

```c
void AssignStmt::typecheck(SymTabScope* s) { 
  Type* lhs = _lvalue->typecheck(s); 
  Type* rhs = _expr->typecheck(s); 
  if (lhs->different(rhs)) { 
    Plzero->typeError("lhs type differs from rhs"); 
  } 
}
```

Type checking if statements

```c
void IfStmt::typecheck(SymTabScope* s) { 
  Type* testType = _test->typecheck(s); 
  if (testType->different(booleanType)) { 
    Plzero->typeError("test not Boolean"); 
  } 
  for (int i = 0; 
       i < _then_stmts->length(); i++) { 
    _then_stmts->fetch(i)->typecheck(s); 
  } 
}
```

(c) Larry Snyder & UW CSE, 2003
Type checking call statements

```c
void CallStmt::typecheck(SymTabScope* s) {
    int i;
    TypeArray* argTypes = new TypeArray;
    for (i = 0; i < _args->length(); i++) {
        Type* argType = _args->fetch(i)->typecheck(s);
        argTypes->add(argType);
    }
    SymTabEntry* ste = s->lookup(_ident);
    if (ste == NULL) {
        Plzero->typeError("undeclared procedure");
        return; // whee! passed all checks!
    }
    Type* procType = ste->type();
    if (! procType->isProcedure()) {
        Plzero->typeError("not a procedure");
    }
    TypeArray* formalTypes = procType->formalTypes();
    if (formalTypes->length() != argTypes->length()) {
        Plzero->typeError("call doesn't match proto");
    }
    for (i = 0; i < formalTypes->length(); i++) {
        if (formalTypes->fetch(i)->different(argTypes->fetch(i))) {
            Plzero->typeError();
        }
    }
    return; // whee! passed all checks!
}
```

Type checking declarations

```c
void VarDecl::typecheck(SymTabScope* s) {
    for (int i = 0; i < _items->length(); i++) {
        _items->fetch(i)->typecheck(s);
    }
}
void VarDeclItem::typecheck(SymTabScope* s) {
    Type* t = _type->typecheck(s);
    VarSTE* varSTE = new VarSTE(_name, t);
    s->enter(varSTE, line);
}
```

```c
void ProcDecl::typecheck(SymTabScope* s) {
    SymTabScope* body_scope = new SymTabScope(s);
    TypeArray* formalTypes = new TypeArray;
    for (int i = 0; i < _formals->length(); i++) {
        FormalDecl* formal = _formals->fetch(i);
        Type* t = formal->typecheck(s, body_scope);
        formalTypes->add(t);
    }
    ProcedureType* procType = new ProcedureType(formalTypes);
    ProcSTE* procSTE = new ProcSTE(_name, procType);
    s->enter(procSTE, line); // add to enclosing scope
    _block->typecheck(body_scope); // check in new scope
}
```

```c
void Block::typecheck(SymTabScope* s) {
    for (int i = 0; i < _decls->length(); i++) {
        _decls->fetch(i)->typecheck(s);
    }
    for (int j = 0; j < _stmts->length(); j++) {
        _stmts->fetch(j)->typecheck(s);
    }
}
```

(c) Larry Snyder & UW CSE, 2003
Type checking

- We’ve covered the basic issues in how to check semantic, type-oriented, properties for the data types and constructs in PL/0 (and some more)

Records

- Records (aka structs) group heterogeneous types into a single, usually named, unit

  ```
  record R = begin
  x : int;
  a : array[10] of bool;
  m : char;
  end record;
  var t : R;
  r.x
  ```

Type checking records

- Need to represent record type, including fields of record
- Need to name user-defined record types
- Need to access fields of record values
- May need to handle unambiguous but not fully qualified names (depending on language definition)

An implementation

- Representing record type using a symbol table for fields
- `class RecordType: public Type {..};`
- `CreateRecordTypeSTE`
- `To typecheck expr.x`
- `Typecheck expr`
  - Error if not record type
  - Lookup x in record type’s symbol table
  - Error if not found
  - Extract and return type of x

Type checking classes & modules

- A class/module is just like a record, except that it contains procedures in addition to simple variables
- So they are already supported by using a symbol table to store record/class/module fields
- Procedures in the class/module can access other fields of the class/module
- Already supported: nest procs in record symbol table
- Inheritance?

Type equivalence

- When is one type equal to another?
  - Implemented in PL/0 with `Type::same function`
  - It’s generally “obvious” for atomic types like `int, string, user-defined types (e.g., point2D vs complex)`
- What about type constructors like arrays?

(c) Larry Snyder & UW CSE, 2003
Equivalence, def I: Structural Eq.
- Two types are structurally equivalent if they have the same structure
  - If atomic types, then obvious
  - If type constructors
    - Same constructor
    - Recursively, equivalent arguments to constructor
  - Implement with recursive same

Equivalence, def II: Name Eq.
- Two types are name equivalent if they came from the same textual occurrence of a type constructor
- Implement with pointer equality of Type instances
- Special case: type synonyms don’t define new types

same & different
- class Type {
  public:
    virtual bool same(Type* t) = 0;
    bool different(Type* t) { return !same(t); }
};
- class IntegerType: public Type {
  public:
    bool same(Type* t) { return t->isInteger(); }
};

Implementing structural equivalence (details)
- Problem: want to dispatch on two arguments, not just receiver
  - That is, choose what method to execute based on more than the class of the receiver
- Why? There’s a symmetry that the OO dispatch approach skews
  - if (lhs->different(rhs)) (...error...)
- Why not: if (different(lhs,rhs)) (...error...)

Multi-methods
- Languages that support dispatching on more than one argument provide multi-methods
- For example, they might look like
  - virtual bool same(type* t1, type* t2)
    (return false);
  - virtual bool same(IntType* t1, IntType* t2)
    (return true);
  - virtual bool same(ProcType* t1, ProcType* t2)
    (return same(t1->args,t2->args));
- Different from static overloading in C++

Overloading: quick reminder
- Overloading arises when the same operator or function is used to represent distinct operations
  - 3 + 4
  - 3.14159 + 2.71828
  - “mark” + “mindy”
- The compiler statically decides which “+” to compile to based on the (type) context

(c) Larry Snyder & UW CSE, 2003
Polymorphism: quick reminder

- Polymorphism is different from overloading
- In overloading the same operator means different things in different contexts
- In polymorphism, the same operator works on different types of data
  - (length "(a b c)" vs. (length "((a) (b c) 3 4)"
  - (sort "(4 1 2)" vs. (sort "(c g a)"
- In polymorphism, the compiler compiles the same code regardless

But C++ has no multi-methods: So we use double dispatching

class Type {
  virtual bool same(Type* t) = 0;
  virtual bool isInteger() { return false; }
  virtual bool isProc() { return false; }
};

class IntegerType : public Type {
  bool same(Type* t) { return t->isInteger();
  bool isInteger() { return true; }
};

Type conversions and coercions

- In C, can explicitly convert data of type float to data of type int (and some other examples)
  - Represent it explicitly as a unary operator
  - Type checking and code generation work as normal
- In C, can also implicitly coerce
  - System must insert unary conversion operators as part of type checking
  - Code generation works as normal

Type casts

- In C, Java (and some others) can explicitly cast an object of one type to another
  - Sometimes a cast means a conversion
    - E.g., casts between numeric types
    - Type-safe, but sometimes entails loss of accuracy
    - Sometimes a cast means just a change of static type without any computation
    - E.g., casts between pointer types
    - Generally NOT type-safe

Safety of casting

- In C, the safety of casts is not checked
  - That is, it’s possible to convert into a representation that is illegal for the new type of data
  - Allows writing of low-level code that’s type-unsafe
  - More often used to work around limitations in C’s static type system
- In Java, downcasts from superclass to subclass include a run-time type check to preserve type safety
  - This is the primary place where Java uses dynamic type checking

Where are we?

- We now know, in principle, how to
  1. take a string of characters
  2. convert it into an AST with associated symbol table
  3. and know that it represents a legal source program (including semantic checks)
- That is the complete set of responsibilities (at a high-level) of the front-end of a compiler

(c) Larry Snyder & UW CSE, 2003
Next...
- ...what to do now that we have this wonderful AST representation
- We'll look mostly at interpreting it or compiling it
  - But you could also analyze it for program properties
  - Or you could “unparse” it to display aspects of the program on the screen for users
- ...

Next lecture
- We'll start looking at the implementation issues in symbol tables
  - For instance, how to efficiently manage references to outer scopes
  - With a particular focus on how PL/0 does it