Semantic Analysis/Checking

Final part of analysis half of compilation

- lexical analysis
- syntactic analysis
- semantic analysis

Afterwards comes synthesis half of compilation

Purposes:

- perform final checking of legality of input program, "missed" by lexical and syntactic checking
  - type checking, `break` stmt in loop, ...
- "understand" program well enough to do synthesis
  - e.g. relate assignments to & references of particular variable

Symbol tables

Key data structure during semantic analysis, code generation

Stores info about names used in program

- declarations add entries to symbol table
- uses of name look up corresponding symbol table entry

Info about names:

- kind of declaration (var, const, proc, ...)
- type
- value, if const
- place allocated in memory (computed later), if var
- whether it's call-by-value or call-by-ref, if formal
- ...

Example

A PL/0 `DeclList`:

```
var x:int;
var q:array[20] of bool;
procedure foo(a:int); begin ... end foo;
const z:int = 10;
```


Nested scopes

How to handle nested scopes?

```
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] := z;
   end bar;
begin
   ...
   while z do
      var z:int, y:int;
      y := z * x;
      ...
   end;
   output := x + y;
end foo;
```

Nested scopes

Can have same name declared in different scopes

Want references to use closest textually-enclosing declaration
- static/lexical scoping, block structure

Simple solution: maintain stack (linked list) of scopes
- stack represents static nesting structure of program
- top of stack = most closely nested

All scopes in program form a tree

Nested scope operations

When enter new scope during semantic analysis/type-checking:
- create a new, empty scope,
  nested in scope on top of stack
- push it on top of scope stack

When encounter declaration:
- add entry to scope on top of stack
- check for duplicates in that scope only
  (allow shadowing of name declared in enclosing scopes)

When encounter use:
- search scopes for declaration, beginning with top of stack
- can find name in any scope

When exit scope:
- pop top scope off stack

Symbol table interface in PL/0

```
class SymTabScope {
 public:
   SymTabScope(SymTabScope* enclosingScope);

   void enter(SymTabEntry* newSymbol);

   SymTabEntry* lookup(char* name);
   SymTabEntry* lookup(char* name,
                       SymTabScope*& retScope);

   ...
};
```


Symbol table entries

```
class SymTabEntry {
 public:
   char* name();
   Type* type();

   virtual bool isConstant();
   virtual bool isVariable();
   virtual bool isFormal();
   virtual bool isProcedure();

   // constants only:
   virtual int value();
   // variables only:
   virtual int offset(SymTabScope* s);

   ...
};

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

Implementation strategies

Abstraction: mapping from names to info

How to implement a map?

Option 1: linked list (or dynamic array) of key/value pairs

- requires only keys that can be compared for equality
- enter time: O(1)
- lookup time: O(*n*) (*n* entries in table)
- space cost: O(*n*)

Option 2: sorted binary search tree

- requires keys that can be ordered
- enter time: O(log *n*) expected, O(*n*) worst case
- lookup time: O(log *n*) expected, O(*n*) worst case
- space cost: O(*n*)

Option 3: hash table

- requires keys that can be hashed well
- enter time: O(1) expected, O(*n*) worst case
- lookup time: O(1) expected, O(*n*) worst case
- space cost: O(*n*)

Summary:

- use hash tables for big mappings,
  binary tree or linked list for small mappings
- ideal: self-reorganizing data structure

Implementing nested scoping

In PL/0: a simple linked list

- each `SymTabScope` points to enclosing `SymTabScope` (`_parent`)
- 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're represented in memory

Type checking uses types to compute whether operations on values will be legal

Types (usually) will guide how compilation proceeds

Taxonomy of types

Basic/atomic types:

- `int`, `bool`, `char`, `real`, `string`, ...
- `enum`(*value*$_1$, ..., *value*$_n$)
- user-defined types: `Stack`, `SymTabScope`, ...

Type constructors:

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

Parameterized types:

- functions returning types
  - `Array<T>`, `HashTable<Key,Value>`, ...

Type synonyms

- give alternative name to existing type
  - e.g. `typedef SymTabScope* SymTabRef;`

Representing types in PL/0

```
class Type {
   virtual bool same(Type* t);
   bool different(Type* t) { return !same(t); }
   ...
};

class IntegerType   : public Type {...};
class BooleanType   : public Type {...};
class ProcedureType : public Type {
   ...
   TypeArray* _formalTypes;
};

IntegerType* integerType;
BooleanType* booleanType;
```


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: can't make guarantees

	static	dynamic
strong		
weak		

Caveats:

- hybrids are common
- mistaken usages are common
- "untyped," "typeless" could mean "dynamic" or "weak"

Type checking

Traverse AST graph recursively, starting at root

- most of work is on bottom-up pass

At each node:

- recursively typecheck subnodes (if any)
- check legality of current node, given types of subnodes
- compute & return result type of current node (if any)

Needs info from enclosing context, too

- need to know types of variables referenced
  $\Rightarrow$ pass down symbol table during traversal
- legality of e.g. `break`, `return` statements
  $\Rightarrow$ pass down whether in loop,
  what result type of function must be

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);
```


Type checking expressions

```
Type* IntegerLiteral::typecheck(SymTabScope* s) {
   // return result type
   return integerType;
}

Type* VarRef::typecheck(SymTabScope* s) {
   SymTabEntry* ste = s->lookup(_ident);

   // check for errors
   if (ste == NULL) {
      Plzero->typeError("undeclared var");
   }
   if (! ste->isConstant() &&
       ! ste->isVariable()) {
      Plzero->typeError("not a var or const");
   }

   // return result type
   return ste->type();
}
```



```
Type* BinOp::typecheck(SymTabScope* s) {
   // check & compute types of subexpressions
   Type* left = _left->typecheck(s);
   Type* right = _right->typecheck(s);

   // check the types of the operands
   switch(_op) {
    case PLUS: case MINUS: ...
      if ( left->different(integerType) ||
           right->different(integerType)) {
         Plzero->typeError("args not ints");
      }
      break;
    case EQL: case NEQ:
      if (left->different(right)) {
         Plzero->typeError("args not same type");
      }
      break;
   }

   // return result type
   switch (_op) {
    case PLUS: case MINUS: case MUL: case DIVIDE:
      return integerType;
    case EQL: case NEQ: ...
      return booleanType;
   }
}
```


Type checking statements

```
void AssignStmt::typecheck(SymTabScope* s) {
   // check & compute types of subexpressions
   Type* lhs = _lvalue->typecheck_lvalue(s);
   Type* rhs = _expr->typecheck(s);

   // check legality of subexpression types
   if (lhs->different(rhs)) {
      Plzero->typeError("lhs and rhs types differ");
   }
}

void IfStmt::typecheck(SymTabScope* s) {
   // check & compute types of subexpressions
   Type* test = _test->typecheck(s);

   // check legality of subexpression types
   if (test->different(booleanType)) {
      Plzero->typeError("test not a boolean");
   }

   // check nested statements
   for (int i = 0; i < _then_stmts->length(); i++) {
      _then_stmts->fetch(i)->typecheck(s);
   }
}
```

```
void CallStmt::typecheck(SymTabScope* s) {
   // check & compute types of subexpressions
   TypeArray* argTypes = new TypeArray;
   for (int i = 0; i < _args->length(); i++) {
      Type* argType = _args->fetch(i)->typecheck(s);
      argTypes->add(argType);
   }
   ProcType* procType = new ProcType(argTypes);

   // check callee procedure
   SymTabEntry* ste = s->lookup(_ident);
   if (ste == NULL) {
      Plzero->typeError("undeclared procedure");
   }
   Type* procType2 = ste->type();

   // check compatibility of actuals and formals
   if (procType->different(procType2)) {
      Plzero->typeError("wrong arg types");
   }
}
```


Type checking declarations

```
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* entry = new VarSTE(_name, t);
   s->enter(entry);
}
```



```
void ConstDecl::typecheck(SymTabScope* s) {
   for (int i = 0; i < _items->length(); i++) {
      _items->fetch(i)->typecheck(s);
   }
}

void ConstDeclItem::typecheck(SymTabScope* s) {
   Type* t = _type->typecheck(s);

   // typecheck initializer
   Type* exprType = _expr->typecheck(s);
   if (t->different(exprType)) {
      Plzero->typeError("init of wrong type");
   }

   // evaluate initializer
   int value = _expr->resolve_constant(s);

   ConstSTE* entry = new ConstSTE(_name, t, value);
   s->enter(entry);
}
```



```
void ProcDecl::typecheck(SymTabScope* s) {
   // create scope for body of procedure
   SymTabScope* body_scope = new SymTabScope(s);

   // enter formals into nested scope,
   // and construct procedure's type
   TypeArray* formalTypes = new TypeArray;
   for (int i = 0; i < _formals->length(); i++) {
      FormalDecl* formal = _formals->fetch(i);
      Type* t = formal->typecheck(body_scope);
      formalTypes->add(t);
   }
   ProcType* procType = new ProcType(formalTypes);

   // add entry for procedure in enclosing scope
   ProcSTE* entry = new ProcSTE(_name, procType);
   s->enter(procSTE);

   // typecheck procedure body last
   _block->typecheck(body_scope);
}

void Block::typecheck(SymTabScope* s) {
   _decls->typecheck(_scope);
   _stmts->typecheck(_scope);
}
```

Type checking records

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

var r:R;

... r.x ...
```

Need:

- represent record type including fields of record
- name user-defined record types
- access fields of record values

An implementation

Represent record type using a symbol table for fields

```
class RecordType: public Type {
   ...
   SymTabScope* _fields;
};
```

To store named record types in symbol tables,
introduce `RecordTypeSTE` symbol table entries

To typecheck `expr.x`:

- typecheck `expr`
  - error if not of record type
- lookup `x` in record type's symbol table
  - error if not found
- extract & return type of `x`

Type checking classes or modules

A class/module is just like a record
that can contain procedures in addition to simple variables

- already supported by using symbol table
  to store record/class/module fields

Procedures in class/module can access other fields of class/
module

- already supported by nesting procedures
  in record symbol table