Semantic analysis

Final part of analysis half of compilation

- lexical analysis
- syntactic analysis
- **semantic analysis**

Afterwards comes synthesis half of compilation

Input: AST

Output: AST

Semantic analysis

Purpose of semantic analysis:

- perform final checking of legality of input program, not done by lexical and syntactic checking

Checks:

- type checking
- relate assignments to & references of particular variables with declarations
- uniqueness checks: e.g., labels, declarations
- same name at beginning and end of a module/procedure declaration
- flow of control checking: `break`

Type checking

Type of a construct is correct for its context

Examples of type checking:

- operands compatible with operator
- function call has correct number & type of arguments
- result of an operation is the correct type
- only index arrays
- only dereference pointers

What we'll cover

Symbol tables: creation & use

Types & type checking

Type equivalence & conversion

Compiling vs. interpreting

Symbol tables

Key data structure during semantic analysis, code generation

Stores information about names used in a program

- keywords can be hardcoded entries
- declarations: **add** entries to symbol table
- uses of name: **look up** symbol table entry

need an organization that grows dynamically

need an organization that reflects scope

Symbol table entries

var	“x”	integer	location (later)
var	“y”	array[20] of bool	location (later)
const	“pair”	integer	2
proc	“foo”	int, float:void	location (later)
formal	“a”	integer: by value	
label	“target”		location (later)
keyword	“if”		
var	“ptr”	pointer to integer	

Implementation strategies

Option 1: **linked list** of key/attribute pairs

- enter time: O(1)
  (assumes no checking to see if already entered)
- 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)

Implementation strategies, cont'd.

Option 3: **hash table**

- requires keys that can be hashed well
- requires good guess of size of table (k)
- enter time: O(1) expected, O(n) worst case
- lookup time: O(1) expected, O(n) worst case
- space cost: O(k+n)

Summary:

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

Nested scopes

How to handle nested scopes?

```
procedure foo(x:int, q: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

Want references to use **closest textually-enclosing declaration**

- static/lexical scoping, block structure

Simple solution: keep stack (linked list) of scopes

- stack represents static nesting structure of program
- top of stack = most closely nested

Used in PL/0

- statically a tree of scopes
- each `SymTabScope` points to enclosing `SymTabScope` (`_parent`)
- maintains "down links," too (`_children`)
- used like a stack during semantic analysis

Nested scope operations

When enter new scope during semantic analysis/type-checking:

- create a new, empty scope
- 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

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) {...};

   // routines to add & lookup ST entries:
   void            enter(SymTabEntry* newSymbol);
   SymTabEntry*    lookup(char* name);
   SymTabEntry*    lookup(char* name,
                          SymTabScope*& retScope);

   // space allocation routines:
   void  allocateSpace();
   int   allocateLocal(int size);
   int   allocateFormal(int size);

   ...
};
```

Symbol table entries

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

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

   // space allocation routine:
   virtual void allocateSpace(SymTabScope* s);

   // 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 { ... };
```


Creating symbol table entries

```
void VarDeclItem::typecheck(SymTabScope* s) {
      ...
      VarSTE* varSTE = new VarSTE(_name, t);
      s->enter(varSTE);
      }

void ConstDeclItem::typecheck(SymTabScope* s) {
      ...
      ConstSTE* constSTE = new ConstSTE
          (_name, t, constant_value);
      s->enter(constSTE);
      }
```

Types

Types are abstractions of values that share common properties

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

Taxonomy of types

Basic types:

- `int`, `bool`, `char`, `real`, `string`, ...
- `void`
- user-defined types: `SymTabScope`, ...

Type constructors:

- `ptr` (*type*)
- `array` (*index-range*, *element-type*)
- `record` ($name_1$:$type_1$, ..., $name_n$:$type_n$)
- `union` ($type_1$, ..., $type_n$)
- `function` (*arg-types*, *result-type*)

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 at compile time
- **dynamic**: checking done during execution

Strong vs. weak typing

- **strong**: guarantees no illegal operations performed
- **weak**: can't make guarantees

	static	dynamic
strong	Ada	Lisp Smalltalk
weak	C Fortran	

Caveats:

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

Bottom-up type checking

Traverse AST graph from leaves up

At each node:

- recursively type check 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
  ⇒ pass down symbol table during traversal
- legality of e.g., `break`, `return` statements
  ⇒ pass down whether in loop, result type of function

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 checking expressions

```
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: ... case LEQ: ...
      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 & rhs types differ");
  }
}
```

Type checking statements

```
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);
  }
}
```


Type checking statements

```
void CallStmt::typecheck(SymTabScope* s) {

  // type check arguments, accumulate list of
  // argument types
  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 & formals
  if (procType2->different(procType)) {
    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);
}
```


Type checking declarations

```
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);

  // type check initializer
  Type* exprType = _expr->typecheck(s);

  // make sure initializer is constant expr
  int value = _expr->resolve_constant(s);

  // make sure rhs matches declared type
  if (t->different(exprType)) {
    Plzero->typeError("init of wrong type");
  }
  ConstSTE* entry =
      new ConstSTE(_name, t, value);
  s->enter(entry);
}
```

Type checking declarations

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

  // enter formals into nested scope
  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);
  }
  // construct procedure's type
  ProcType* procType =
      new ProcType(formalTypes);

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

  // type check procedure body
  _block->typecheck(body_scope);
}
```


Starting out

```
int TPlzero::main2(int argc, char** argv) {
  ...
  typeCheckPhase(); ...

void TPlzero::typeCheckPhase() {
    module->typecheck(NULL); ...
  // no enclosing symbol table


void ModuleDecl::typecheck(SymTabScope* s) {
    // create new scope for body of module
    SymTabScope* body_scope =
     new SymTabScope(s);
    // type check body of module
    _block->typecheck(body_scope);
}

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

Extensions

1) adding `else` to `if`

2) adding `for` statements

3) adding `break` statements

4) adding constant expressions

5) adding arrays

6) adding call-by-reference

7) adding `return` statements

Consult the project description!

Type checking records.

For the type ‘record’:

- represent record type & fields of record
- represent public vs. private nature of fields

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

Need to be able to:

- give names to user-defined record types
- access fields of record values

```
var r:R;

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

An implementation

Represent record type using a symbol table for fields

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

Add `RecordSTE` symbol table entries for user-defined types (**`R`**)

Add `FieldSTE` symbol table entries for fields (**`r.x`**)

For public vs. private, add boolean flag to `SymTabEntry`

```
class FieldSTE : public SymTabEntry {
public:
  FieldSTE(char* name, Type* t, bool p) :
        SymTabEntry(name, t, p) {}
};
```


An implementation

To type check **`r.x`**:

- type check **`r`**
- check it's a record
- lookup **`x`** in **`r`**'s symbol table
- check that it's public,
  or that current scope is nested in record (private)
- extract & return type of **`x`**

var	"r"	record	pointer to ST for fields

field	"x"	integer	public	location (later)
field	"a"	array[10] of bool	public	location (later)
field	"m"	char	private	location (later)

Type equivalence

Type checking often involves knowing when two types are equal

- implemented in PL/0 with `Type::same` function

When is one type equal to another?

"Obvious" for basic types like `int`, `char`, `string`

What about type constructors like arrays? (pl0)

```
var a1     :array[10] of int;
var a2,a3  :array[10] of int;

var a4     :array[20] of int;
var a5     :array[10] of bool;
```


Structural vs. name equivalence

Structural equivalence:

two types are equal if they have same structure

- basic types
- type constructors:
  - same constructor
  - structurally equivalent arguments to constructor, recursively

- example (Pascal)

```
type ar1 = array [1..10] of integer;
type ar2 = array [1..10] of integer;
var myArray : ar1;
var yourArray : ar2;
myArray := yourArray;
```

- implement with recursive implementation of `same`

Structural vs. name equivalence

Name equivalence:

two types are equal if they came from the same **textual** occurrence of a type constructor

- example (Ada)

```
type LIST_10 is array (1..10) of integer;
C, D : LIST_10
E : LIST_10
```

- requires that all types have a name
  - defined names

```
type celsius is FLOAT;
type fahrenheit is FLOAT;
```

  - anonymous types

```
A: array (Integer range 1..10) of integer;
(type Anonymous1 is array
  (Integer range 1..10) of Integer;)
```

- each declaration has a different name

```
B: array (Integer range 1..10) of integer;
A: array (Integer range 1..10) of integer;
```


Structural vs. name equivalence

```
type count is int;
type index is int;
sheep, toTen : count;
blessings : count;
a : index;
b : index;
```

Structural vs. name equivalence

```
TYPE t1 = ARRAY [1..10] OF INTEGER,
     t2 = t1;
TYPE t3 = ARRAY [1..10] OF INTEGER;

VAR  x: t1;
     y: t2;
     z: t3;
     w: ARRAY [1..10] OF INTEGER;
```


Type conversions and coercions

Why needed:

- types have different representations
- different machine instructions used for different types

Type conversions and coercions

Implicit conversion (coercion)

- type system does the conversion automatically
- system must insert unary conversion operators as part of type checking
- done where code does not “make sense” for type checking or code generation

- example (C)

  `i = j + 2.1416`

  j converted to real; real truncated to int

\+ programmer not have to code it

\- only coerce if no loss in precision: “widening”

- e.g., an object of type `int` to one of type `float`

Type conversions and coercions

Explicit conversion

- conversion stated in the code
- using unary operators (“casting”) or built-in functions

\+ provides more flexibility in kinds of conversions

\- programmer has to do it

- example of functions (Modula-2)

  `i := TRUNC (FLOAT (j) + 2.1416)`

  j converted to real; real truncated to int

- example of functions (Fortran)

  `INT(r)`: real, double, complex to integer

  `REAL(i)`: integer to real

  `DBLE(i)`: integer to double

  `CMPLX(i)`: integer to complex

- example of casting (C)

```
float pi;
int pentium;
... (int)pi / pentium;
```

Overloading

Symbol has different meaning depending on the context

- e.g., same operation on different types
  - + for integer add
  - + for floating point add
- Different implementations for each type

Why do it? to avoid proliferation of names

Overloading is resolved when a unique type is determined

- **semantic** rules (e.g., look at types of operands for +)
- **context** determines the type
- example that uses both (Ada)

```
function "*" (i,j:integer) return integer;
function "*" (i,j:integer) return complex;
function "*" (x,y:complex) return complex;
```

means that (3*5)*complexValue is complex

means that (3*5)*integerValue is integer

- bottom-up type checking
- choose a unique operation top-down

Polymorphic functions

Non-polymorphic function: arguments have fixed type

Polymorphic functions: arguments can have different types

- why do it? support for ADTs
- one implementation
- binding between code & type done dynamically
- example (C): pointer operator `&`
- example (ML)

```
fun first(x,y) = x
... first (2,3)...
... first (2.0,3.0)...
... first ([1,2], [3,4])...
```

Example (C++)

`typecheck` is **overloaded**
(different implementation depending on the context)

If you consider the receiver an argument, than `typecheck` is a **polymorphic function**
(its argument has different types)

```
virtual Type* typecheck(SymTabScope* s) {
   Plzero->fatal("need to implement");
   return NULL; }

Type* VarRef::typecheck(SymTabScope* s) {
   SymTabEntry* ste = s->lookup(_ident);
      if (ste == NULL) {
      Plzero->typeError("undeclared var"); }
   if (! ste->isConstant() &&
       ! ste->isVariable()) {
      Plzero->typeError("not a var or const"); }
   return ste->type(); }
```


Implementing a language

Given type-checked AST program representation:

- can generate target program that is then run separately (**compiler**)
- can interpret AST directly, carry out operations given data input (**interpreter**)

Interpreters

Simulate the program:

Create data structures to represent run-time program state

- activation record for each called procedure
  - environment to store local variable bindings
  - pointer to calling activation record (**dynamic link**) for procedure return
  - pointer to lexically-enclosing activation record/ environment (**static link**) for variable lookups

Interpretation loop that evaluates the AST and executes code to carry out the operations

Pros and cons of interpretation

+ simple conceptually, easy to implement
+ good programming environment for program development & debugging
+ some machine independence

- slow to execute
  - evaluation overhead vs. direct machine instructions
  - no optimizations across AST nodes
  - program text is reexamined & analyzed
  - variable lookup vs. registers or direct access
  - data structures for values vs. machine registers & stack

Compilation

Divide interpreter run-time into two parts:

- compile-time
- run-time

Compile-time does preprocessing

- perform analysis of source code & synthesis of target code at compile-time once
- produce an equivalent but more efficient program in machine language that gets run many times

Only advantage over interpreters: faster running programs

Compile-time processing

Decide representation of run-time data values

Decide where data will be stored

- format of in-memory data structures (e.g. records, arrays)
- registers
- format of stack frames
- global memory

Do optimizations across instructions

Generate machine code to do basic operations

- just like interpreting expression, except generate code that will evaluate it later