CSE401: Semantic Analysis
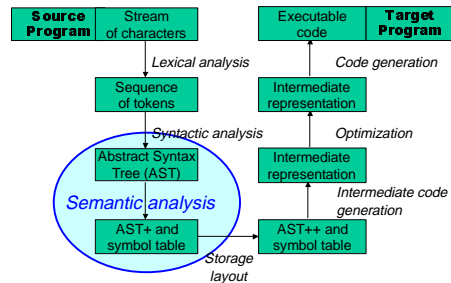
Larry Ruzzo

Spring 2002

Slides by Chambers, Eggers, Notkin, Ruzzo, and others
© W.L. Ruzzo and UW CSE, 1994-2002

Prototype compiler structure

2

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

3

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

4

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…

5

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;
```

6

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
}
```

More soon

7

`SymTab` subclasses

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

8

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

9

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?

10

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
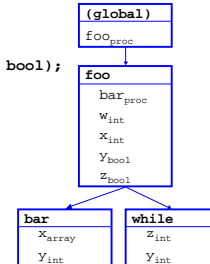
11

Nested scopes: Dynamic

12

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

13

Tree of symbol tables

```
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;
```

(global)
foo$_{proc}$

foo
bar$_{proc}$
w$_{int}$
x$_{int}$
y$_{bool}$
z$_{bool}$

bar
x$_{array}$
y$_{int}$

while
z$_{int}$
y$_{int}$

14

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

15

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)
  - New scope becomes "current"
- 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

Stack-like

16

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);
  …
}
```

17

Implementing nested scopes

- Each scope (instance of `SymTabScope`) keeps a pointer to its enclosing `SymTabScope` (`_parent`)
- Each scope maintains "down links", too (`_children`, so we can walk the whole tree)

18

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…

19

Symbol tables: Complexity

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

20

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?

21

Symbol tables: 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

22

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

23

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

24

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 $type_1 \times ... \times type_n \rightarrow$ result-type

25

Parameterized types

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

26

Type synonyms

Give alternative name to existing type
- `typedef SymTabScope* SymTabReg`

27

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 + 4.0
3 + x	3 + 'x'
3[x]	x[3]
3 + TRUE	x.y->z*

28

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 "weak"

29

Type weaknesses in C/C++

```
extern myfunc(double*);
main() {
  int i=42, j=0, *ip=&i;
  double x=3.14, y[10];
  scanf("%d %f", &i, &j);
  x     = (double) i;
  x     = (double*) ip;
  (*ip)  = 1;
  (++ip) = 1;
  y[11]  = 1;
  myfunc(&x);
}                    main.c
```

```
myfunc(int *kp){
  char c='1';
  union{
    int i;
    double x;
  } huh;

  c = sqrt(c);
  huh.x = 42.0;
  huh.i += 1;
  *kp = huh.i;
}                    myfunc.c
```

30

More on C++ type system

```
Stmt* sp;
IfStmt* isp;
isp  = new IfStmt(…);
sp = isp;
sp = (Stmt*) isp;
…
isp = (IfStmt*) sp;
sp = (isp -> _then_stmts->fetch(14)) ;
//Better:
if(isp = dynamic_cast<IfStmt*> sp) {
  sp = isp -> _then_stmts->fetch(14);
}
```

upcast – always safe

downcast – safe? dynamic check? (Java would)

31

Fill in with real languages

	Statically typed	Dynamically typed
Strong typing		
Weak typing		

32

Type checking

- Assume we have an AST for the source program
  - It is syntactically correct
  - The symbol table has been computed
- Does it meet the type constraints of the language?
  - Ex: `a := 3 * b + fork(c + 3.14159)`
    – What are the types 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`?
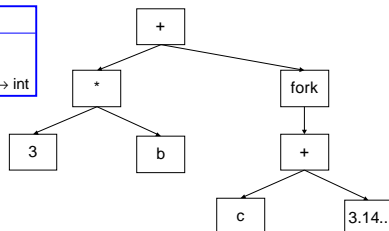   – What happens when `fork`'s result is added to `3 * b`?

33

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

34

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

Symtab
b: int
c: float
fork: float → int

```
        +
      /   \
     *     fork
    / \      |
   3   b     +
            / \
           c   3.14..
```

35

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.

36

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;     // predefined instances
BooleanType* booleanType;
```

37

PL/0 type checking: 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);
```

38

Type checking PL/0 expressions

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

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

39

Type checking var references

```
Type* VarRef::typecheck(SymTabScope* s) {
    SymTabEntry* ste = s->lookup(_ident);
    if (ste == NULL) {
      char* errormsg = new char[errormsgbuffsize];
      sprintf(errormsg,
          "undeclared var \"%s\" referenced", _ident);
      Plzero->typeError(errormsg, line);
    }
    if (! ste->isConstant() &&
        ! ste->isVariable()) {
      char* errormsg = new char[errormsgbuffsize];
      sprintf(errormsg,"\"%s\" not const or var",_ident);
      Plzero->typeError(errormsg, line);
    }
    return ste->type();
}
```

40

Type checking operators

```
Type* BinOp::typecheck(SymTabScope* s) {
    Type* left  = _left->typecheck(s);
    Type* right = _right->typecheck(s);

    switch(_op) {
      case PLUS:case MINUS:case MUL: 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;

      default:
        Plzero->fatal("unexpected BINOP");
    }
```

41

```
    switch (_op) {
      case PLUS:case MINUS:case MUL:case DIVIDE:
        return integerType;

      case EQL:case NEQ:case LSS:
      case LEQ:case GTR:case GEQ:
        return booleanType;

      default:
        Plzero->fatal("unexpected BINOP");
        return NULL;  // not actually executed
    }
}
```

42

Type checking assignments

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

43

Type checking if statements

```
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);
      }
  }
```

44

Type checking call statements

```
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");
   }
```

45

```
  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;        // whew! passed all checks!
}
```

46

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

47

```
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* type = _expr->typecheck(s);
  Value* constant_value = _expr->resolve_constant(s);
  if (t->different(type)) {
    Plzero->typeError(…);
  }

  ConstSTE* constSTE =
     new ConstSTE(_name, t, constant_value);
  s->enter(constSTE, line);
}
```

48

```
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);            [Continued]

  ProcSTE* procSTE = new ProcSTE(_name, procType);
  s->enter(procSTE, line);  // add to enclosing scope

  _block->typecheck(body_scope); // check in new scope
}
```
49

```
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);
  }
}
```
50

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)
- But there are other features in languages richer than PL/0, and we'll look at some of them today

51

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
```
52

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)
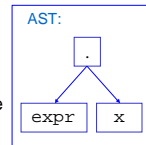
53

An implementation

- Representing record type using a symbol table for fields
  - class RecordType: public Type {..};
  - Create RecordTypeSTE
- 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

AST:

```
      .
     / \
  expr   x
```

54

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?

55

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?

```
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
var a6    : array[0:9] of int;
```

56

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`

57

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

58

`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(); }
   …
};
```

59

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…}`

60

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++

61

Overloading: quick reminder

- Overloading arises when the same operator or function is used to represent distinct operations
  - `3 + 4`
  - `3.14159 + 2.71828`
  - `"mork" + "mindy"`
- The compiler statically decides which "+" to compile to based on the (type) context

62

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

63

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;}
};
```

64

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

65

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

66

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

67

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

68

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
  - …

69