CSE401: Semantic Analysis (B)

David Notkin
Autumn 2000

Today

- Type checking in PL/0
- Miscellaneous issues in type checking

Type checking

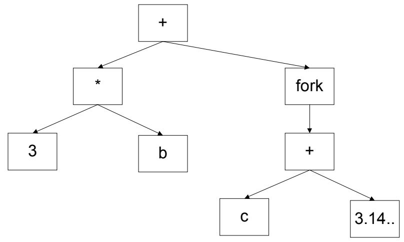
- Assume we have an AST for the source program
  - It is syntactically correct
  - The symbol table has been computed
- Now we need to check to see if it meets the type constraints of the language
  - Ex: `a := 3 * b + fork(c + 3.14159)`
  - What is the type of `a`, `b`, and `c`? What type does `fork` 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`?

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

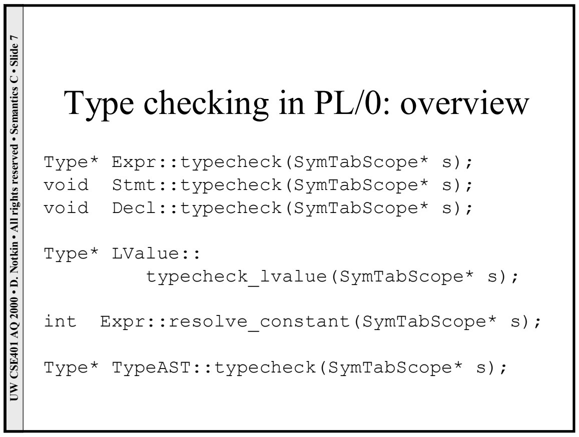
Example:

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



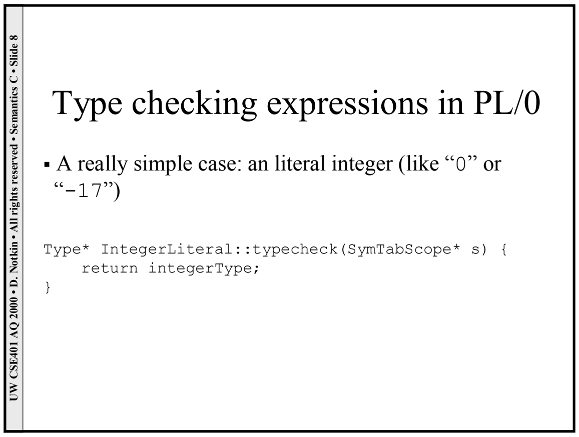
Representing types in PL/0

```
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; …
```



Type checking in PL/0: overview

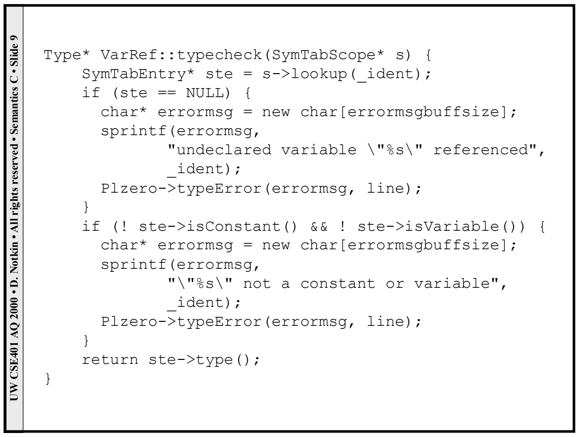
```
Type* Expr::typecheck(SymTabScope* s);
void  Stmt::typecheck(SymTabScope* s);
void  Decl::typecheck(SymTabScope* s);

Type* LValue::
         typecheck_lvalue(SymTabScope* s);

int  Expr::resolve_constant(SymTabScope* s);

Type* TypeAST::typecheck(SymTabScope* s);
```



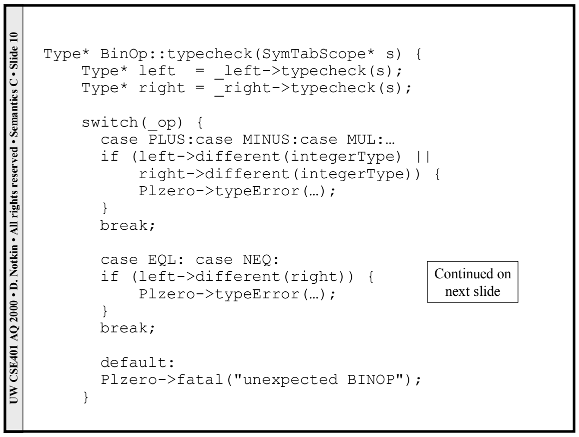
Type checking expressions in PL/0

- A really simple case: an literal integer (like “`0`” or “`-17`”)

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

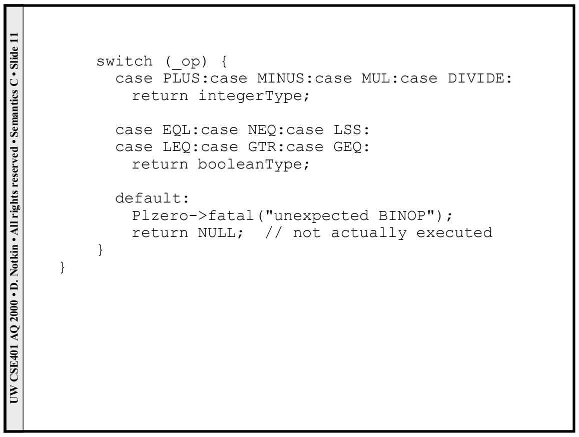


```
Type* VarRef::typecheck(SymTabScope* s) {
    SymTabEntry* ste = s->lookup(_ident);
    if (ste == NULL) {
      char* errormsg = new char[errormsgbuffsize];
      sprintf(errormsg,
             "undeclared variable \"%s\" referenced",
             _ident);
      Plzero->typeError(errormsg, line);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

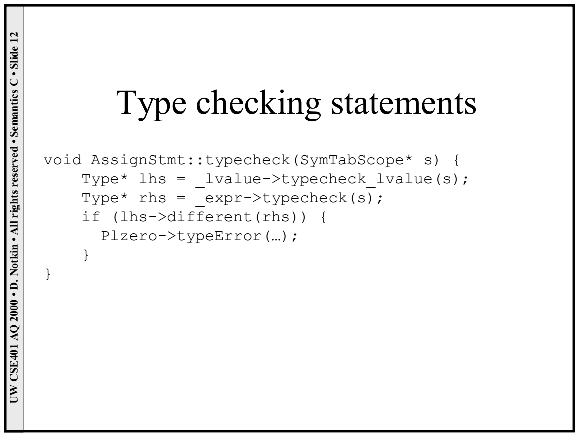


```
Type* BinOp::typecheck(SymTabScope* s) {
    Type* left  = _left->typecheck(s);
    Type* right = _right->typecheck(s);

    switch(_op) {
      case PLUS:case MINUS:case MUL:…
      if (left->different(integerType) ||
          right->different(integerType)) {
          Plzero->typeError(…);
      }
      break;

      case EQL: case NEQ:
      if (left->different(right)) {
          Plzero->typeError(…);
      }
      break;

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

Continued on next slide

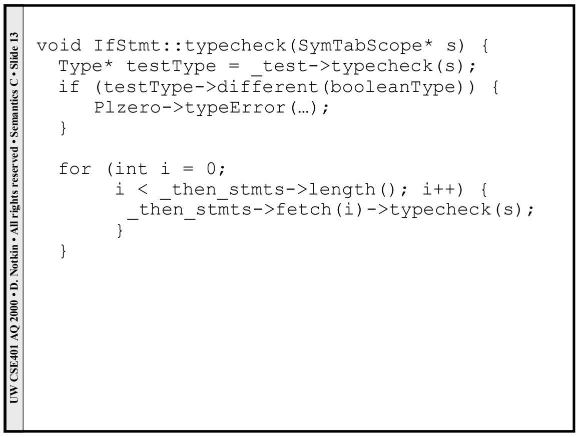


```
    switch (_op) {
      case PLUS:case MINUS:case MUL:case DIVIDE:
        return integerType;

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

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

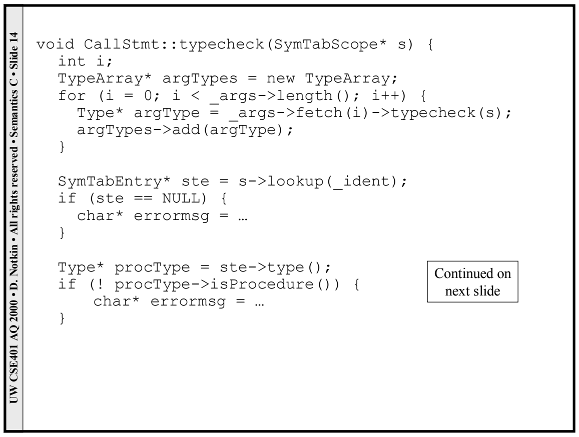


Type checking statements

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

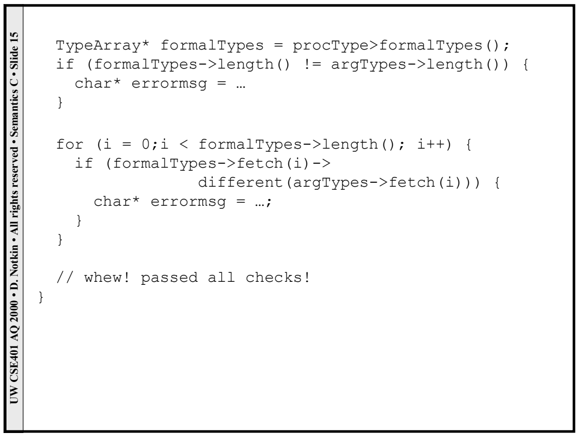


```
void IfStmt::typecheck(SymTabScope* s) {
  Type* testType = _test->typecheck(s);
  if (testType->different(booleanType)) {
     Plzero->typeError(…);
  }

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

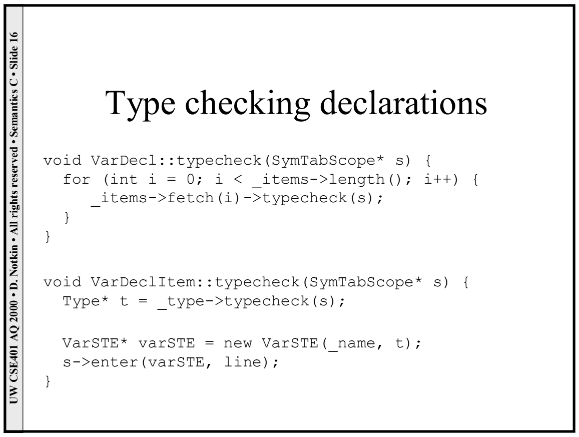


```
void CallStmt::typecheck(SymTabScope* s) {
  int i;
  TypeArray* argTypes = new TypeArray;
  for (i = 0; i < _args->length(); i++) {
    Type* argType = _args->fetch(i)->typecheck(s);
    argTypes->add(argType);
  }

  SymTabEntry* ste = s->lookup(_ident);
  if (ste == NULL) {
    char* errormsg = …
  }

  Type* procType = ste->type();
  if (! procType->isProcedure()) {
      char* errormsg = …
  }
```

Continued on next slide

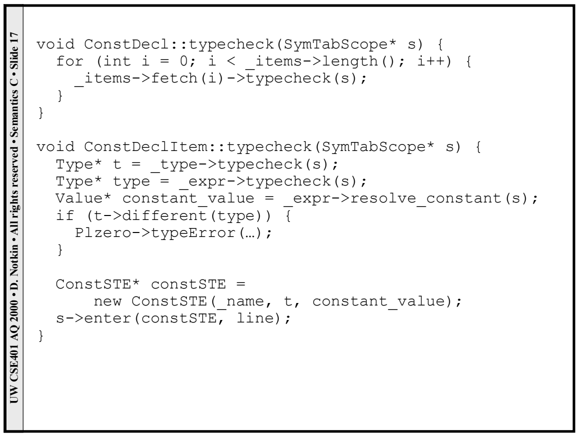


```
  TypeArray* formalTypes = procType>formalTypes();
  if (formalTypes->length() != argTypes->length()) {
    char* errormsg = …
  }

  for (i = 0;i < formalTypes->length(); i++) {
    if (formalTypes->fetch(i)->
                 different(argTypes->fetch(i))) {
      char* errormsg = …;
    }
  }

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

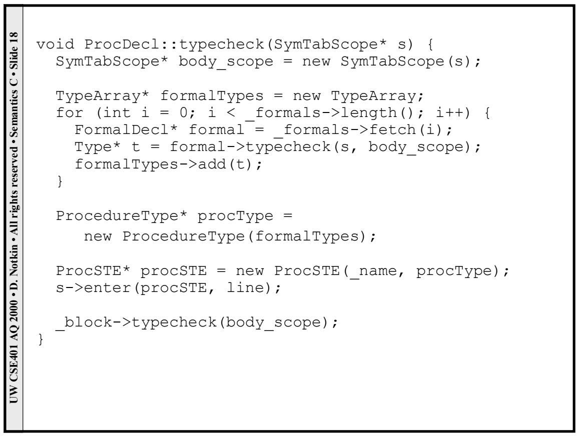


Type checking declarations

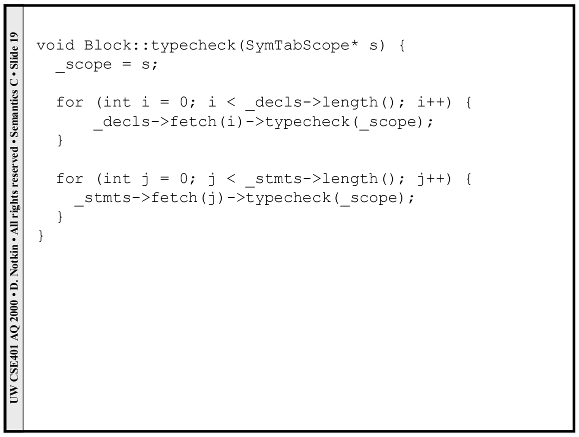
```
void VarDecl::typecheck(SymTabScope* s) {
  for (int i = 0; i < _items->length(); i++) {
     _items->fetch(i)->typecheck(s);
  }
}

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

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



```
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);
}
```



```
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);

  _block->typecheck(body_scope);
}
```



```
void Block::typecheck(SymTabScope* s) {
  _scope = s;

  for (int i = 0; i < _decls->length(); i++) {
      _decls->fetch(i)->typecheck(_scope);
  }

  for (int j = 0; j < _stmts->length(); j++) {
    _stmts->fetch(j)->typecheck(_scope);
  }
}
```