Discussion 1

CSE 131

introduction

See course page for the lab hour schedule.

We are available at our lab hours, or by appointment. Feel free to email us.

overview

- purpose of our discussions
- starter code
- STOs and types
- error reporting
- functions

requisite prior knowledge

- object oriented design
- inheritance
- polymorphism
- some C
- some assembly (for project 2)

compilation phases

- lexical analysis
    - parsing of tokens - not a major point of this course
    - Lexer.java
- syntactic and semantic analysis
    - project 1 focuses on semantics (does the code make sense?)
    - rc.cup and MyParser.java
- code generation
    - project 2

starter code

- /home/solaris/ieng9/cs131s/public/starterCode/
- look through the files and become familiar with them
- the GETTING_STARTED and CUP_Overview files are helpful

important files

rc.cup contains the parser’s rules and action code

```
Designator2 ::=
     Designator2:_1 T_DOT T_ID:_3
     {:
          RESULT = ((MyParser) parser).DoDesignator2_Dot (_1, _3);
     :}
|    Designator2:_1 T_LBRACKET ExprList T_RBRACKET
     {:
          RESULT = ((MyParser) parser).DoDesignator2_Array (_1);
     :}
;
```

cup syntax

```
Designator2 ::=
    Designator2:LABEL T_DOT T_ID:LABEL
    {:
        ACTION CODE GOES HERE
    :}
|   Designator2:_1 T_LBRACKET ExprList T_RBRACKET
    {:
        // RESULT is the “return” value of the rule
        RESULT = _1;
    :}
;
```

important files

MyParser.java contains methods for semantic analysis

```
STO DoDesignator3_ID (String strID)
{
     STO sto;
     if ((sto = m_symtab.access (strID)) == null)
     {
          m_nNumErrors++;
          m_errors.print (Formatter.toString(ErrorMsg.undeclared_id, strID));
          sto = new ErrorSTO (strID);
     }

     return (sto);
}
```

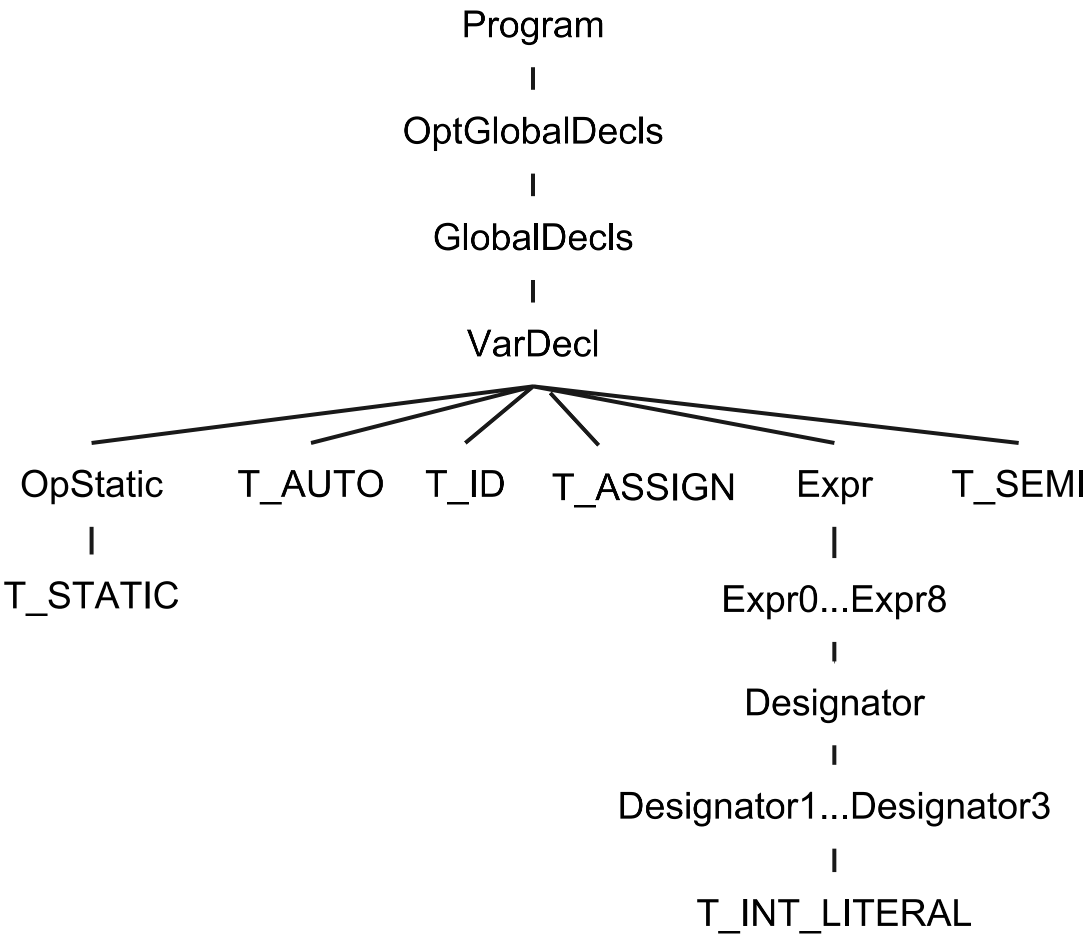
important files

SymbolTable.java contains functions that support scopes

(Contains one of the problems that will be fixed in phase 0)

parse tree example

```
auto x = 2;
```



Program

OptGlobalDecls

GlobalDecls

VarDecl

OpStatic

T_AUTO

T_ID

T_ASSIGN

Expr

T_SEMI

T_STATIC

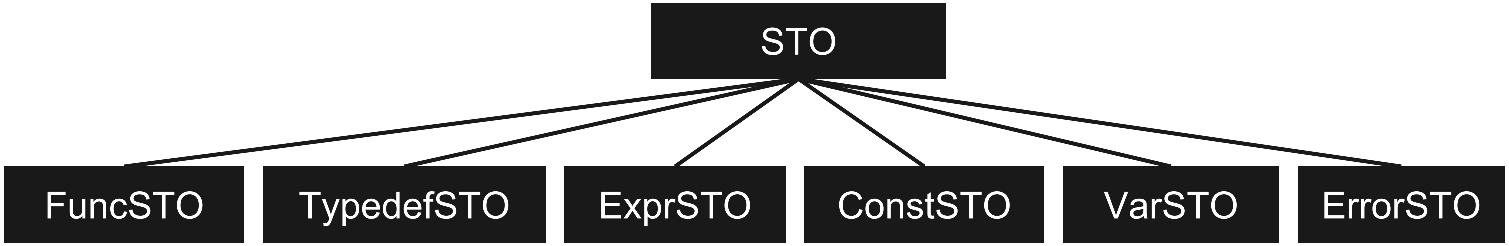
Expr0...Expr8

Designator

Designator1...Designator3

T_INT_LITERAL

sto hierarchy



- you can change this around
- look at the methods in each and how they are overloaded (e.g., isVar(), isConst())

sto hierarchy

- all STOs have fields to indicate modifiability and addressability
    - whenever you create an STO instance, make sure to set these appropriately
- ConstSTO includes a value field
    - used for constant folding (more on this later)

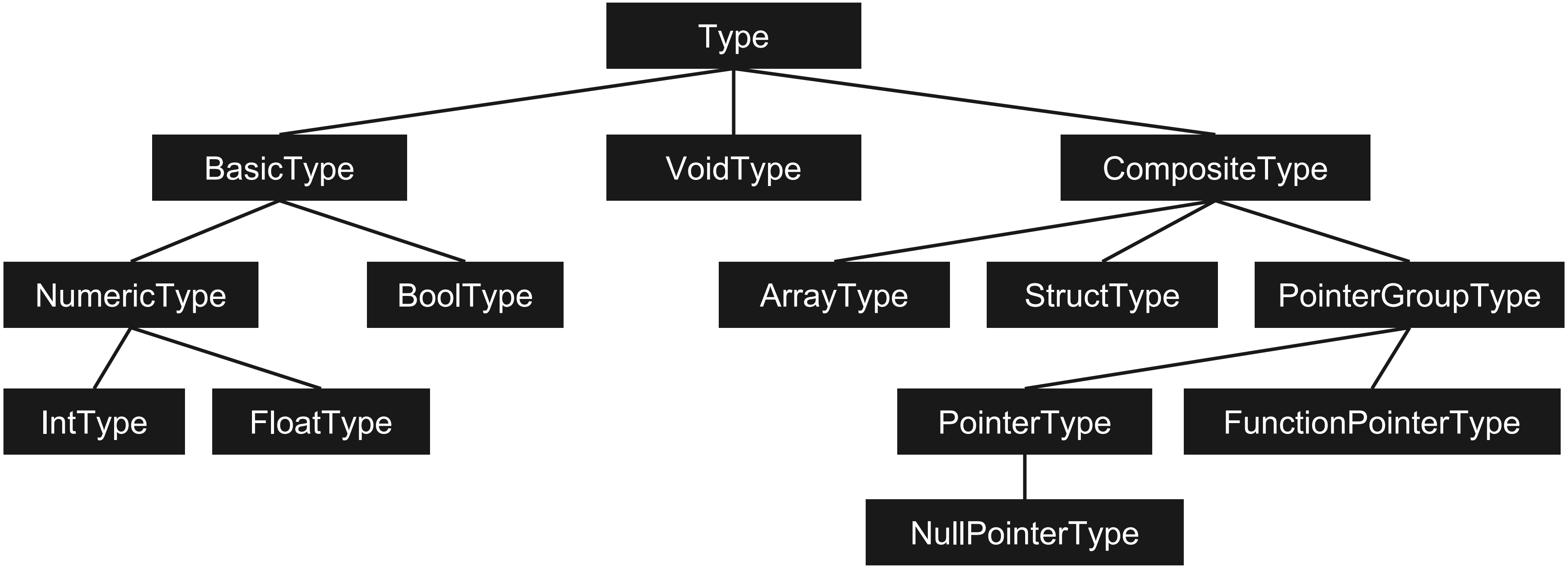
types

```
Type ::=
     SubType OptModifierList OptArrayDef
|    T_FUNCPTR T_COLON ReturnType T_LPAREN OptParamList:_3 T_RPAREN
     ;

SubType ::=
     QualIdent
|    BasicType
     ;

BasicType ::=
     T_INT
|    T_FLOAT
|    T_BOOL
|    T_CHAR
;
```

types

- need to create objects for
    - basic types
    - array types
    - struct types (only done in structdef)
    - pointer types
    - function pointer types
- how can they be organized to make our lives easier?
- what methods and fields should we provide within each?

possible type hierarchy



useful methods on types

- look at how the STO files are written
- consider making methods like
  - isNumeric()
  - isFloat()
  - isInt()
  - isArray()
  - etc
- even better, use Java’s instanceof operator
  - e.g. obj instanceof NumericType

useful methods on types

- all types would benefit from methods like:
  - isAssignableTo(Type t) - coercible type (int -> float)
  - isEquivalentTo(Type t) - same type
- some types will need to store more info
  - ArrayType may need dimensions
  - StructType may need a list of member fields
  - size of the type for sizeof

setting types

- must ensure all STOs have a type field
  - make sure this field is set properly when the type becomes known
- what changes need to be made to the rc.cup and MyParser.java files?

type setting example

```
VarDecl ::= OptStatic UndecoratedType IdentListWOptInit:_3 T_SEMI
{:
     ((MyParser)parser).DoVarDecl(_3);
:}
```

- we want to incorporate the type, so we pass it to the MyParser method

type setting example

now, in MyParser.java

```
void DoVarDecl (Vector lstIDs, Type t)
{
     for (int i = 0; i < lstIDs.size (); i++)
     {
          String id = (String) lstIDs.elementAt (i);
          if (m_symtab.accessLocal (id) != null)
          {
               m_nNumErrors++;
               m_errors.print(Formatter.toString(ErrorMsg.redeclared_id, id));
          }

          VarSTO sto = new VarSTO (id);
          // Add code here to set sto’s type field
          m_symtab.insert (sto);
     }
}
```

type checking

- now that our STOs have types, how do we check them?

type checking example

- consider the following code:

```
int x;
float y;
function : void main() {
     x = 5; // ok
     y = x + 12.5; // ok
     x = y; // error
}
```

- let’s focus on y = x + 12.5

type checking example

Currently rc.cup has:

```
Expr7 ::= Expr7:_1 AddOp:_2 Expr8:_3
{:
     RESULT = _1;
:}
```

- What needs to be done?
  - based on AddOp (+, -) we need to check the types of _1 and _3
  - then create a new STO (an ExprSTO) to return as the result

type checking example

- getting the type out of an STO?
- have STO a and want to check type equivalence to some STO b?
  - a.getType().isEquivalentTo(b.getType())

important themes

- don’t just throw code into the files
- think about the current problem at hand
- think about upcoming tasks
- try to make your code as general as possible

- project 2 (assembly generation) will be based off project 1, so the more forethought you put into project 1 the easier project 2 will be

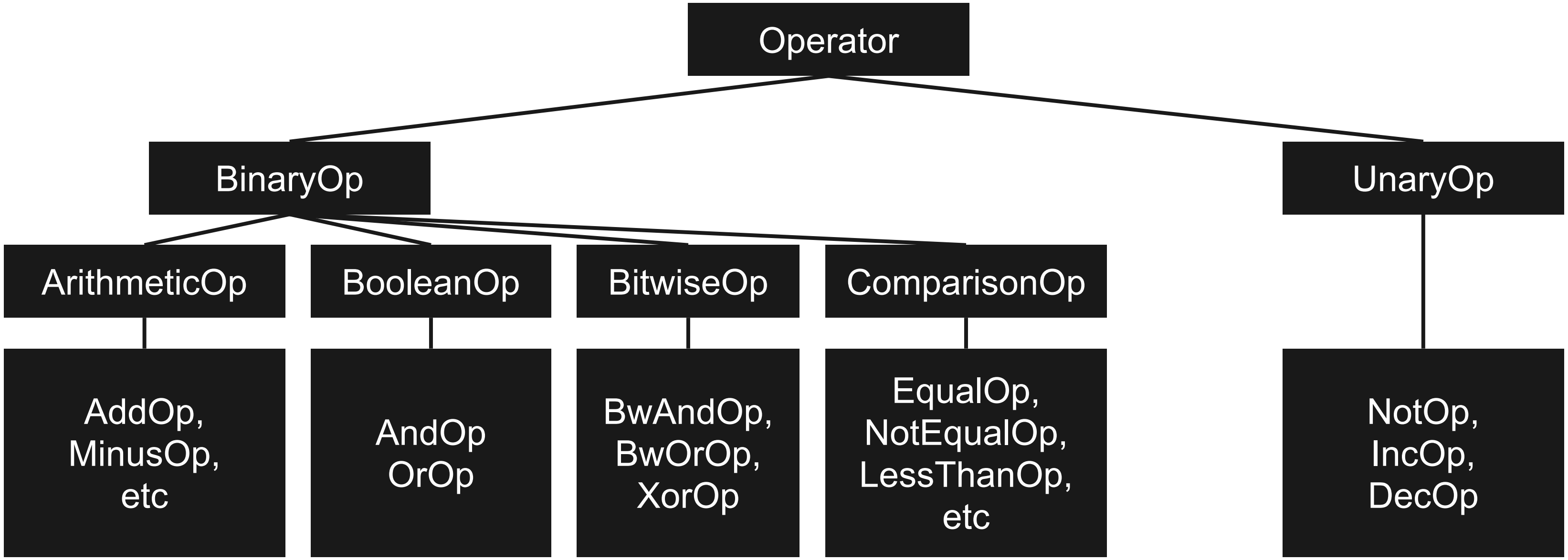
an idea

- define a new function inside MyParser to check expressions
- define Operator classes to assist type checking

```
// in MyParser.java
STO DoBinaryExpr(STO a, Operator o, STO b) {
     STO result = o.checkOperands(a, b);
     if (result instanceof ErrorSTO) {
          // do stuff
     }

     return result ;
}
```

operator hierarchy

- probably the most important hierarchy of this assignment, one possible setup:



an idea

In each operator class, we can do something like this:

```
STO checkOperands(STO a, STO b) {
    aType = a.getType();
    bType = b.getType();
    if (!(aType instanceof NumericType) || !(bType  instanceof NumericType)) {
        // error
        return new ErrorSTO(...);
    } else if (aType instanceof IntType && bType instanceof IntType) {
        // return ExprSTO of int type
    } else {
        // return ExprSTO of float type
    }
}
```

error reporting

- now that we can check types, we will find errors
- once found, they need to be printed out
- use only the provided error messages in ErrorMsg.java

error reporting

- only report the FIRST error found in each statement
- if you want to see the line number where the error occurred (for debugging only) use “make debug”

ErrorSTO

- looks just like any other STO
- when you find an error, make the result an ErrorSTO
    - this provides a signal to higher level rules that an error happened below them somewhere
    - use that signal to know when to stop further checks within in the statement

functions

- some fundamental points
    - we are writing a static translator, not an interpreter
    - once we finish the function declaration, including the body, we’re done with it
        - don’t need to remember code in the body
        - in project 2 you’ll spit out assembly code for it
    - function calls will boil down to an assembly “call foo” sort of instruction

functions

- for project 1
  - check the function call against the function declaration to ensure argument symmetry
  - do type checking on the statements in the body of the function
  - check the return logic of the function, including return by reference
  - allow function overloading (extra credit)

FuncSTO

- store some information about the function
    - return type
        - separate from the full type of the function itself, which is actually a function pointer type
    - flag for return by reference
    - parameter information
        - total number of parameters
        - type of each parameter, including pass-by-reference or not

function definition

```
FuncDef ::=
    T_FUNCTION T_COLON ReturnType OptRef T_ID:_2
        {:
            ((MyParser) parser).DoFuncDecl_1(_2);
        :}
    T_LPAREN OptParamList:_3 T_RPAREN
        {:
            ((MyParser) parser).DoFormalParams(_3);
        :}
    T_LBRACE OptStmtList T_RBRACE
        {:
            ((MyParser) parser).DoFuncDecl_2();
        :}
;
```

function definition

```
void DoFuncDecl_1(String id) {
     if (m_symtab.accessLocal(id) != null) {
          m_nNumErrors++;
          m_errors.print(Formatter.toString(ErrorMsg.redeclared_id, id));
     }

     FuncSTO sto = new FuncSTO(id);
     m_symtab.insert(sto);       // Inserted into current scope

     m_symtab.openScope();       // New scope opened
     m_symtab.setFunc(sto);      // Current function we’re in is set
}
```

function definition

```
void DoFuncDecl_2() {
     m_symtab.closeScope();      // Close scope (pops top scope off)
     m_symtab.setFunc(null); // Says we’re back in outer scope
}
```

function definition

```
function : bool foo(float a, float b, float &c) {
     bool x;
     x = a > b;
     x = (a + c) <= 2;
     return x;
}
```

- In this example
  - we need to make a FuncSTO with
    - name foo
    - return type boolean
    - parameter count
    - parameters: value float, value float, reference float

function definition

```
function : bool foo(float a, float b, float &c) {
     bool x;
     x = a > b;
     x = (a + c) <= 2;
     return x;
}
```

- further, we need to insert VarSTOs for a, b, and c into the symbol table so the code in the body can use them

function calls

- now that we have a type checked FuncSTO in the symbol table, we are ready to call it

foo(1, 2, 3.3)

- given the call above, we’d have an error since 3.3 is not addressable (for the ref param)

function calls

- when we call a function
  - get the matching FuncSTO from the symbol table and check its parameters with the arguments at the callsite
- consider making a vector of some object to hold the parameter info
- important design choices
- remember to think ahead about function overloading (extra credit)

function return types

- inside the function body
  - type of the return expression needs to be checked against the declared return type of the function
- at the callsite
  - the function call behaves just like any other ExprSTO
  - the return type becomes the type of the expression
- functions are the only thing that can have a void type
  - void isn't equivalent or assignable to anything ever (including itself)

function return types

- return by value
  - in the function, the return expression just has to be assignable to the function’s return type
  - at the callsite, the resulting expression is an rval
- return by reference
  - in the function, the return expression has to have an equivalent type AND must be a modifiable lval
  - at the callsite, the resulting expression results in a modifiable lval

next steps

- find a partner
  - maybe set up a unix group (see broadcast message)
- set up source control
  - git, svn, etc
  - if using a hosted solution like github or bitbucket, make sure the repo is private
- consider using Eclipse
- understand the starter code
  - look through all the files
- do phase 0 and phase 1

next steps

- come to lab hours
  - we’re here for you
- check piazza
  - often other people have the same questions as you and they may already be answered