Compilers [Spring 2020]

Programming Assignment V

Objectives

1. To learn basic typing rules (i.e., *static semantics*) for an object-oriented language with inheritance and subtyping.
2. To implement an enhanced symbol table for Diminished Java (DJ) programs.
3. To implement a type checker for DJ programs.

Due Date: Sunday, April 12, 2020 (at 11:59pm).

Machine Details

Complete this assignment by yourself on the cselx##.csee.usf.edu computers. You are responsible for ensuring that your programs compile and execute properly on these machines.

Assignment Description

This assignment asks you to extend your dj2dism compiler with type checking (i.e., semantic analysis). The type checker will rely on symbol-table data structures.

Begin by downloading and studying the header files for the symbol-table module (in file *symtbl.h*) and the type-checking module (in file *typecheck.h*). These files are posted at http://www.cse.usf.edu/~ligatti/compilers/20/as5.

Implement the `typecheckProgram` function (declared in *typecheck.h*) in a new file called *typecheck.c*. For extra credit, you may also implement the `setupSymbolTables` function (declared in *symtbl.h*) in a new file called *symtbl.c*.

If you are completing this assignment by extending your Assignment IV implementation (as is recommended), you will also have to make a few minor modifications to your *dj.y*:

1. The *symtbl.h* and *typecheck.h* header files need to be included.
2. The main function in *dj.y*, which is the main function of the entire compiler, needs to invoke `setupSymbolTables` and `typecheckProgram` so that the compiler actually performs the type checking.
3. Once you are confident that your compiler produces correct ASTs, you will no longer want your compiler to print ASTs during normal operation (users generally do not want to see the ASTs their compilers build). Therefore, remove the call to `printAST` from *dj.y* (possibly, e.g., by setting a `DEBUG_PARSE` flag to 0).

Notes on Typing DJ Programs (beyond those given in the *Definition of DJ* handout):

- A list of expressions has the same type as the final expression in the list.
- A DJ method *M* may "redeclare" variable names locally or in its parameter that are already defined as class variables. In this case, the local/parameter declaration *punctures the scope* of the class variable, so any use of the variable in *M* refers to the local/parameter rather than the class variable. However, a variable-expression block *B* may not redeclare variables that have already been declared in *B* (or that have the same name as a parameter that can be used in *B*).
- The expression `null` has type "any object", which is a subtype of every class.

- The expression `new C()` has type C.
- An equality expression E1==E2 is well typed (with type *bool*) exactly when (1) E1 has valid type T1, (2) E2 has valid type T2, and (3) either T1 is a subtype of T2 or T2 is a subtype of T1.
- An if-then-else expression is well typed exactly when its if-expression has *bool* type and either (1) the then- and else-expression-lists both have *nat* type, or (2) the then- and else-expression-lists both have *bool* type, or (3) the then- and else-expression-lists both have an object type. In cases (1) and (2), the type of the entire if-then-else will match the type of the branches. In case (3), the type of the entire if-then-else expression is the *join* of the types of the then- and else-expression-lists. For example, the expression `if(true) {null;} else{new Object();}` has type Object because the join of "any-object" type and Object is Object.
- A for-loop expression is well typed (with a *nat* type) exactly when its loop-initialization subexpression, loop-conclusion subexpression, and loop-body expression-list are all well typed, and its loop-test subexpression has *bool* type.
- A method that expects a parameter of type C may be passed any parameter that is a subtype of C.
- A method *M*, declared to return a value of type *T*, may return any value whose type is a subtype of *T*. In any case, the method's return type is still considered to be its declared return type *T*.
- The assignment expression ID=E (where ID is an identifier and E is an expression) is well typed (with whatever type ID has) exactly when (1) ID is a well-typed variable and (2) E's type is a subtype of ID's type. Assignment expressions of the form E.ID=E are type checked similarly.
- An instanceof expression E instance ID is well typed (with a *bool* type) exactly when E is any kind of object type and its ID is a valid class name.
- A use of the keyword *this* must appear inside a declaration of some class C, in which case *this* has type C.
- The class hierarchy declared in a DJ program may not contain cycles (e.g., we cannot have C1 a subclass of C2, C2 a subclass of C3, and C3 a subclass of C1). Similarly, no class may be its own superclass.
- Overriding methods' parameter *names* may differ from those of overridden methods.
- The main block of a DJ program must be well typed (though it may have any type).
- Of course, all variables must be declared prior to use.

The examples at http://www.cse.usf.edu/~ligatti/compilers/20/as1/dj/examples/bad/ are intended to illustrate all the high-level sorts of typing errors that are possible in DJ. Although more typing errors are possible, the additional possibilities should be simple variations of errors illustrated in the posted examples. (E.g., *bad19.dj* illustrates an error in which the second operand in a conjunction expression has non-*bool* type; obviously your type checker needs to ensure that both operands have *bool* type.)

Hints on Implementing the Enhanced Symbol Tables (in *symtbl.c*)

You may find it helpful to organize your *symtbl.c* such that it implements the following functions (in addition to the `setupSymbolTables` function):

```
/* Return the number of children an AST node has.
   Note: Children with NULL data are not counted. */
int countChildren(ASTree *parent)

/* Returns the number for a given type name:
    -5 if type doesn't exist, -2 for bool, -1 for nat, 0 for Object,
     1 for first class declared in program,
     2 for 2nd class declared in program, etc.
     (-3 and -4 are reserved for "any-object" and "no-object" types) */
int typeNameToNumber(char *typeName)

/*  Build a variable symbol table from a VAR_DECL_LIST or a
     STATIC_VAR_DECL_LIST. */
VarDecl *buildVarST(ASTree *decls)

/* Build a method symbol table from a METHOD_DECL_LIST AST. */
MethodDecl *buildMethodST(ASTree *decls)

/* Set the global count of the number of declared classes, and
   set the global classesST (an array of ClassDecl structs). */
void buildClassEntries()

/* Set the mainExprs, numMainBlockLocals, and mainBlockST global
   variables. */
void buildMainST()
```

Hints on Implementing the Type Checker (in *typecheck.c*)

You may find it helpful to organize your *typecheck.c* such that it implements the following functions (in addition to the `typecheckProgram` function):

```
/* Returns nonzero iff sub is a subtype of super */
int isSubtype(int sub, int super)

/* Return the join (i.e., least upper bound) of two types.
   Assumes the parameters are joinable, i.e., if t1 or t2 are < 1 then:
     either t1 is a subtype of t2 (in which case the join is t2)
     or t2 is a subtype of t1 (in which case the join is t1). */
int join(int t1, int t2)

/* Returns the type of the expression AST in the given context.  Also
   sets t->staticClassNum, t->isMemberStaticVar, and t->staticMemberNum
   attributes as needed.
   If classContainingExpr < 0 then this expression is in the main block
   of the program; otherwise the expression is in the given class. */
int typeExpr(ASTree *t, int classContainingExpr,
             int methodContainingExpr)

/* Returns the type of the EXPR_LIST AST in the given context. */
int typeExprs(ASTree *t, int classContainingExprs,
              int methodContainingExprs)
```

More Hints

My *symtbl.c* file is 289 lines of code (72 of which are comments/whitespace), while my *typecheck.c* file is 718 lines of code (71 of which are comments/whitespace). This is a

relatively challenging assignment, requiring you to write several hundred lines of code. Please budget your time accordingly.

Many examples of valid and invalid DJ files, on which you can test your type checker, are posted at http://www.cse.usf.edu/~ligatti/compilers/20/as1/dj/examples. A complete solution to this assignment will detect an error in every one of the "bad" example DJ programs and no errors in any of the "good" example DJ programs. As usual, we will grade your type checker on DJ programs that have not been distributed to the class.

Grading

Your grade on this assignment is determined by the level to which you implement the desired functionality:

- *Level I*: `typecheckProgram` correctly type checks all DJ programs that (1) declare no classes and (2) declare no local variables (hence you never even need to use the symbol tables). Please note that at this level you can *only* assume that DJ programs input to your type checker declare no classes and no variables; input programs may still contain expressions with identifiers, the *new* keyword, the *null* keyword, etc. Expect to write about 250 lines of code for this level (or about 200 without counting comments and whitespace).
- *Level II*: `typecheckProgram` correctly type checks all DJ programs that declare no classes. For this level, expect to write about 50 lines of code beyond that of Level I.
- *Level III*: `typecheckProgram` correctly type checks all DJ programs (with the `typeExpr` function properly setting `staticClassNum`, `isMemberStaticVar`, and `staticMemberNum` AST-node attributes, which should be used in Assignment VI).

Undergraduate students will earn: 75% credit for reaching Level I, 85% credit for reaching Level II, and 105% credit for reaching Level III.

Graduate students will earn: 65% credit for reaching Level I, 75% credit for reaching Level II, and 100% credit for reaching Level III.

All students will earn +5% extra credit for correctly implementing `setupSymbolTables` in a file called *symtbl.c*.

Compilation of the Type Checker

To compile your type checker from scratch, use the following sequence of commands.

```
> flex dj.l
> bison -v dj.y
> sed -i '/extern YYSTYPE yylval/d' dj.tab.c
> gcc dj.tab.c ast.c symtbl.c typecheck.c -o dj-tc
```

Alternatively, you can download a working version of the lexer and parser as object-code file *dj.tab.o* at http://www.cse.usf.edu/~ligatti/compilers/20/as5. Also at that URL you will find object-code files *ast.o* (the AST module) and *symtbl.o* (which can be used if you decide not to implement *symtbl.c*). These object-code files are executable on the C4 Linux machines.

If you have downloaded *dj.tab.o*, *ast.o*, and *symtbl.o* (in addition to the header files *ast.h*, *symtbl.h*, and *typecheck.h*), compile your type checker with:

```
> gcc dj.tab.o ast.o symtbl.o typecheck.c –o dj-tc
```

Example Executions

If the input DJ program is valid, your type checker should output nothing.

```
> ./dj-tc good1.dj
```

Otherwise, your type checker must report at least one error. Assuming the error is a syntactic or semantic (typing) error, your type checker must exit after reporting the error. *All typing errors must be reported with (1) reasonably accurate and helpful error messages and (2) the line numbers on which the errors occur.* For example:

```
> ./dj-tc bad3.dj
Semantic analysis error on line 7:
   Variable declared multiple times (here and in a superclass)
> ./dj-tc bad9.dj
Semantic analysis error on line 3:
   Invalid declared return type
> ./dj-tc bad14.dj
Semantic analysis error on line 9:
   non-nat type in printNat
> ./dj-tc bad44.dj
Semantic analysis error on line 6:
   reference to 'this' outside of a class
```

Submission Notes

- Type the following pledge as an initial comment in your *typecheck.c* and (optional) *symtbl.c* files: "I pledge my Honor that I have not cheated, and will not cheat, on this assignment." Type your name after the pledge. Not including this pledge will lower your grade 50%.
- Upload and submit your *typecheck.c* and (optional) *symtbl.c* files on Canvas. Upload and submit each file by itself; do not zip them into one.
- ***When submitting your code in Canvas, include with your submission a comment indicating at which level we should grade your assignment: I, II, or III. If you do not indicate a level, we will grade your submission as a Level-III submission.***
- You may submit your assignment in Canvas as many times as you like; we will grade your latest submission.
- For every day that your assignment is late (up to 2 days), your grade reduces 10%.
- To make it easier for our teaching assistant to read and evaluate your code, use spaces rather than tabs in your code and avoid long lines of code (I try to limit lines to 80 characters in width).
- Your programs will be graded on both correctness and style, so include good comments, well-chosen variable names, etc. in your programs.
- For full credit, your submissions must compile on the cselx machines without warnings and without errors.