Compilers [Spring 2021]

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 4, 2021 (at 11:59pm).

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/21/as5.

Implement the `setupSymbolTables` method (declared in *symtbl.h*) in a new file called *symtbl.c* and implement the `typecheckProgram` method (declared in *typecheck.h*) in a new file called *typecheck.c*.

If you are completing this assignment by extending your Assignment IV implementation (as is recommended), you'll 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 method in *dj.y*, which is the main method of the entire compiler, needs to invoke `setupSymbolTables` and `typecheckProgram` so that the compiler actually performs the type checking.
3. Once you're confident that your compiler produces correct ASTs, you'll no longer want your compiler to print ASTs during normal operation (users generally don't want to see which 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 *nat*) 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 *nat* type and either (1) the then- and else-expression-lists both have *nat* type, or (2) the then- and else-expression-lists both have an object type. In case (1), the type of the entire if-then-else is *nat*. In case (2), 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 while-loop expression is well typed (with a *nat* type) exactly when its loop-test expression has *nat* type and its loop-body expression-list is well typed.
- 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 analogously.
- An assert expression is well typed (with *nat* type) exactly when its subexpression has *nat* type.
- 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.
- Final classes may not have subclasses, and final methods may not be overridden.
- 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/21/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 disjunction expression has non-*nat* type; obviously your type checker needs to ensure that both operands in a disjunction have *nat* type.)

Hints

My *symtbl.c* file is 281 lines of code (68 of which are comments/whitespace), while my *typecheck.c* file is 607 lines of code (65 of which are comments/whitespace; many others are checks that pointers about to be dereferenced are non-null). This is a relatively challenging assignment. Please budget your time accordingly.

Many DJ examples are at http://www.cse.usf.edu/~ligatti/compilers/21/as1/dj/examples/. A complete solution to this assignment will detect an error in every one of the "bad"

example programs and no errors in any of the "good" example programs. As usual, we'll grade your type checker on DJ programs that haven't been distributed to the class.

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/21/as5. Also at that URL you'll find the object-code file *ast.o* (the AST module). These object-code files are executable on the C4 Linux machines.

If you've downloaded *dj.tab.o* and *ast.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.c 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 *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 *symtbl.c* files on Canvas. Please upload and submit each file by itself; do not zip them into one.
- You may submit your assignment in Canvas as many times as you like; we'll grade your latest submission.
- To make it easier for our teaching assistant to read and evaluate your code, please 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.