Semantic Analysis
Deliverable Report C

Assignment 3

Team 2

Design Document

Instructor:	**Prof. Dave Wortman**
Course:	**CSC488S Compilers & Interpreters**
Due Date:	**February 26th 2015 1:00 PM**
Team Members:	**Benson Quach (g0quachb)**
	Eric Chen (g3chencm)
	Eric Snyder (g4snyder)
	Francesco Gramano (g2graman)
	Nicholas Dujay (c4dujayn)
	Winston Yeung (g2yeungw)

Contributions

Nicholas Dujay (c4dujayn)

1. Modified the parser to generate a complete AST with Benson
2. Designed and implemented the visitor pattern with Benson, modifying all AST Node class files and modularizing it so all focus would be in Semantics java class
3. Implemented Error handling with Benson, modified to include line & column number and printing out invalid syntax
4. Completed major and minor scope & general table scope searching through linked list, adding SymbolTable, SymbolTableEntry java class files, and modifying MajorScope
5. Implemented Expressions & Expression Type Checking & modified/fixed various semantic checks S25,26,27,28,35,36,37,38,39,11,12,50,51,52,53
6. Contributed to passing/failing test case writeups
7. Pair programmed with Benson on virtually all tasks in each of our lists

Benson Quach (g0quachb)

1. Modified the parser to generate a complete AST with Nicholas, Annotated cup file with line & column number, and capturing/declaring AST Node information
2. Designed and implemented the visitor pattern with Nicholas, modifying all AST Node class files and modularizing it so all focus would be in Semantics java class
3. Implemented Error handling with Nicholas, including error handling function and respective calls
4. Handled Design Document formatting and write up, included README.A3 & README.tests, created test case files containing respective semantic operator descriptions for the team to write test cases
5. Implemented Expressions Types & Expression Type Checking & modified/fixed various semantic checks S20,21,23,24,30,31,32,34,11,12,50,51,52,53
6. Contributed to passing/failing test case writeups
7. Pair programmed with Nicholas on virtually all tasks in each of our lists

Eric Chen (g3chencm)

1. Implemented Declarations & Functions, procedures and arguments semantic checks S10,13,14,15,16,17,18,19,46,47,48,40,41,42,43,44,45 with Winston
2. Contributed to passing/failing test case writeups, handled majority failing cases
3. Helped fix bugs found in the implementation

Winston Yeung (g2yeungw)

1. Implemented Declarations & Functions, procedures and arguments semantic checks S10,11,12,13,14,15,16,17,18,19,46,47,48,40,41,42,43,44,45 with Eric Chen
2. Contributed to passing/failing test case writeups, handled majority failing cases
3. Helped fix bugs found in the implementation

Eric Snyder (g4snyder)

1. Implemented Scopes and Program & Statement Checking semantic checks S00,01,03,04,05,06,07,08,09,50,51,52,53 with Francesco
2. Wrote first level major scoping with Francesco
3. Contributed to passing/failing test case writeups

Francesco Gramano (g2graman)

1. Implemented Scopes and Program & Statement Checking semantic checks S00,01,03,04,05,06,07,08,09 with Eric Snyder
2. Wrote first level major scoping with Eric Snyder

Marking Scheme for Assignment 3

Design and implementation [30%]

How well structured and organized is their code.

- We implemented the visitor pattern which requires modification of nearly every AST node in the assignment, all or rather most logic for this assignment can be found in the Semantics java class and cup file
- For a brief rundown of the visitor pattern, please refer to Visitor Pattern Rundown section below

Good internal comments counts toward the documentation mark. Is their compiler complete, does it build a complete AST?

- To build a complete AST we took the completed cup file that was provided in the course and modified it to capture all respective values. This includes line and column numbers to pass to the respective AST node objects declared.

Does it implement all of the semantic checks?

- All semantic checks/logic can be found in the Semantics.java file with the respective comments of each semantic check commented with //S# immediately above

Documentation [25%]

Did they document the symbol table design ?

- The symbol table was designed using a linked list where the head of the list is the most inner scope and the tail is the furthest outer scope such as the main body.
- Each node respectively contained a hashmap of variable to respective symbol table entry objects that contained the respective fields being:

```
String varname,
Type type,
Kind kind; // can be either array, variable, function, procedure
AST node;
```

Document basic symbol table operations and scope handling. Should document how they deal with major (main program, function, procedure) scopes, minor scopes ({} scopes embedded in major scopes)

- To handle major scopes we create a pseudo stack of scopes and search from the top of the stack to the bottom. Whenever a new scope is created, it is pushed on top of this stack, and when a scope is exited it is popped off. It is implemented as a linked list to prevent having to pop all the scopes off then pushing them back on.
- To handle minor scopes we simply place the new identifiers and respective symbol table entry into the respective scope table

How do they handle function/procedure parameters?

- The parser.cup file will generate a parameter list of ScalarDecls and then the Semantic checker will just visit the list of ScalarDecls and create the declarations in the scope of the newly created routine.

How do they handle array declarations?

- ArrayDeclParts are visited by the Semantics class which then actually declares each ArrayDeclPart individually.

How do they handle AST building design?

- The AST was constructed using the provided CUP file, specifying and structuring accordingly to it and the nodes (java objects) were respectively declared accordingly through the provided framework given to us.
- All we had to modify was the CUP file to capture and instantiate AST nodes

AST building - changes/additions to the AST should be well documented. Additions for handling source tracking, symbol table links, type tracking.

- Source tracking or line location was tracked by modifying the cup file annotations and ASTBase java file with line and column numbers
    - The cup file was modified using a setLocation(line#, column#) format to the respective AST Node
- Symbol table link tracking was handled using Symbol Table Entry objects keeping track of their respective AST Nodes through a field. Refer to SymbolTableEntry for more information.
- Type tracking was handled by modifying the respective super class for expressions. Declarations already contained a field to handle type tracking. Expn contained a field for type and had functions that allowed comparison, getting, and setting. Refer to Expn for more information.

Semantic Analysis design? Should describe all of the mechanisms that they added to the skeleton to implement semantic analysis.e.g. type tracking, scope tracking, etc. What method did they use to process the AST?

- The overall design as mentioned above is visitor pattern for semantic analysis design, type tracking was dealt with by repurposing the AST Node with new fields and methods, scope tracking was dealt with using a linked list, as mentioned above.
- All semantic checks can be found within the Semantic java file

Their testing? They should document the testing that they did, what test cases they used, What parts of the compiler they tested.

- In regards to specific test cases refer to the index section at the end of this document.

Testing [30%]

They should test symbol table and semantic analysis with correct programs and programs containing errors. Just running their A1 programs through their compiler isn't nearly enough for full testing marks. How thoroughly did they test their symbol table? How thoroughly did they test their AST building? How thoroughly did they test their semantic analysis?

- Both passing and failing test cases were included in our testing folder for the three sections mentioned. Please refer to the Index section and our tests folder for detail.
- We also tested against the A2 test cases that were provided on the discussion board, with of course slightly modified to have semantically valid declarations in the passing folder

Visitor Pattern Rundown

We have implemented the Semantic class as an ASTVisitor. Semantics.java will traverse the AST. Each AST node must implement an accept method that defines how the inputted visitor will visit the node exactly.

For example, the BinaryExpn has two children: Left and Right

```
// Below statement will traverse left and right, so left.accept(visitor), right.accept(visitor)
BinaryExpn.accept(visitor)
```

```
BinaryExpn {
    Expn left;
    Expn right;

    public Boolean accept(ASTVisitor<Boolean> visitor) {
        left.accept(visitor); // check the left expression
        right.accept(visitor); // check the right expression
        return visitor.visit(this); // Semantics.visit(BinaryExpn expn) will get called
                                    // check this expression after left and right have been checked
    }
}
```

Another example, the ReturnStmt has a single child: expn

ReturnStmt.accept(visitor) will traverse its only child, so expn.accept(visitor)

```
ReturnStmt {
    Expn value;

    public Boolean accept(ASTVisitor<Boolean> visitor) {
        value.accept(visitor); // check the expression
        return visitor.visit(this); // Semantics.visit(ReturnStmt stmt) will get called
                                    // check this statement after the value has been checked
    }
}
```

ASTVisitor also expects you to return a value. Semantics expects all the values returned to be a Boolean. If visit(node) returns true, then node is semantically correct. IF visit(node) returns false, then `node` has a semantic error.

Following the function calls

In Main.java, theres the following lines of code:

Main.java

```
ASTVisitor<Boolean> visitor = new Semantics();
programAST.accept(visitor);
```

The programAST.accept method is defined in compiler488.ast.stmt.Scope
(which is a parent class of Program)

compiler488/ast/stmt/Scope.java

```
public Boolean accept(ASTVisitor<Boolean> visitor) {
    visitor.visit(this); // This line will call Semantics.visit(Scope scope);
    // since the type of `this` is Scope in Scope.java
    return this.body.accept(visitor);
    //this is defined in ASTList.java
}
```

compiler488/semantics/Semantics.java

since we are passing in semantics as a Semantics.visit(Scope scope)
// do some semantic actions for this scope, for example S06
the visit(ASTList list) defined in here will visit all the elements of the list correctly

Index of Test Cases

As per the specifications this section is to describe in one sentence/comment on each test case we performed. In the files with more than one test case, the passing cases, we also comment each of the cases. In terms of testing, we wrote test cases in the language of this course that should pass or fail with respect to the three sections mentioned.

Test Cases:

- ***Symbol Table (Can be found in the ScopesAndPrograms.488 test file)***
    a. Major Scoping (begin end, function, procedure)
    b. Minor Scoping (if statement, while loop, loop end)
- ***AST Building***
    a. We don't have time to manually construct and check AST Building, but we are reassured that it works since Semantic Analysis testing works properly
- ***Semantic Analysis (All other test case files)***

  Declarations
    a. S46 Check that lower bound is<=upper bound.

  Statement Checking
    b. S50 Check that exit statement is directly inside a loop.
    c. S51 Check that return is inside a function
    d. S52 Check that return statement is inside a procedure.
    e. S53 Check that a function body contains at least one return statement.

  Expression Type Checking
    f. S30 Check that type of expression is boolean.
    g. S31 Check that type of expression or variable is integer.
    h. S32 Check that left and right operand expressions are the same type.
    i. S34 Check that variable and expression in assignment are the same type.
    j. S35 Check that expression type matches the return type of enclosing function.
    k. S36 Check that type of argument expression matches type of corresponding formal parameter.
    l. S37 Check that identifier has been declared as a scalar variable.
    m. S38 Check that identifier has been declared as an array.
    n. S39 Check that identifier has been declared as a parameter.

  Functions, procedures and arguments
    o. S40 Check that identifier has been declared as a function.
    p. S41 Check that identifier has been declared as a procedure.
    q. S42 Check that the function or procedure has no parameters.
    r. S43 Check that the number of arguments is equal to the number of formal parameters.

What to submit / How to submit Assignment 3

1) Submit tar ball named csc488h.A3.XY.tar (where XY is your team number).
If more than one tar ball is submitted by a team we will use the most
recent one.

The tar ball should contain:

a) a complete compilable source tree for your Assignment 3 compiler.
We will compile your compiler from source and run it on our test cases.
We will do

```
ant distclean        # just to be sure
ant gettools
ant dist
```

and will use the compiler dist/compiler488.jar to test your software.

We don't need/want compiled files, so you should do "ant distclean" before
you tar your source tree.

We do not need/want the javadoc for your compiler. We can generate
it if required. "ant distclean" will remove any existing javadoc

[The truly paranoid will verify that their compiler
can be recompiled from the directory tree in the tar ball.
Paranoia is recommended by many software engineers including the instructor.]