Milestone 3

Haiyang He

March 22, 2022

Required information

- name: Haiyang He
- UCID: 30067349
- email: haiyang.he@ucalgary.ca
- gitlab repo `https://gitlab.cpsc.ucalgary.ca/haiyang.he/cpsc411`

Briefly say what you're most proud of in this milestone's code and why I am proud of the following things:

- Instead of using a linked list for my look up in my symbol table, I wrote up a hash map with open addressing and it uses FNV-1 hashing algorithm [1]. It turns out that once you have a hashing function, the open addressing way is very easy to implement. It was not the perfect hash map, because I mod the hash by the array size, which does not give uniformly distributed array indices. The only problem with open addressing is we need to reallocate the entire array when it is full, but that is easily detectable if we store a size parameter (we will not delete in the hash map in this assignment anyways, so we have no problems).

  I also implemented my own stack (which is simply a linked list), and I try to free the stack everytime it goes out of scope, which gives some memory management.

- For the symbol table, instead of reallocating a node and storing that pointer inside the hashmap, I just stored the AST node pointer directly. This enables us to deal less with the memory management, which is good in C.

  Then the problem is, how do we distinguish whether if the pointer is a variable declaration node or a function declaration node? I could just wrap it around inside a struct, but there is a neat pointer trick for this: we can steal the high bit (MSB) of the pointer and use that to determine the 2 cases. This is justified in 2 ways: one is that it is unlikely that this will be run on a machine with $2^6 3 - 1$ bits (about 281Tb) of RAM, and second, some java virtual machine uses this technique for its garbage collector (and they steal a lot more bits than we did!), for example: `https://dinfuehr.github.io/blog/a-first-look-into-zgc/`.

- For the traversals, I only did 2 passes: one pass is to get the top level declarations, and the second pass does the rest. For this compiler at least, I don't think it is necessary to go multiple passes, since the information we want to pass down and up are very limited. It is not very ugly to do either, since the information we need to pass down is only triggered at very specific stages (for example, a break statement), and it was quite clear what every recursion needed to check.

- To detect if the main function is called, from the previous milestones I did not store this information inside the function itself. Luckily, I fixed this problem by just adding a specific return type of main inside bison, and then we are good to go.

  I also need to extract out the identifier case inside the expression to a struct called `ExpID`, which just stores the identifier and the symbol table pointer. These changes were not hard to make, and I think partially it is because my AST was extensible, even though it follows the grammar closely.

Briefly say what in this milestone's code needs the most improvement and why One thing that I should improve on is the traversal when an error is detected, especially for the assignment expression case. Later in the testing, we can see that every subexpression inside a statement expression will error, which is not the ideal error message the user likes to see.

Another thing I can improve on is the number of scopes inside the stack. Since we cannot have local declaration in an outermost block, whenever we enter a function it suffices to push one scope on the stack, and that is it. Right now for every block I will push to the stack, but we cannot declare anything inside the inner blocks, which is a waste of allocation. I think it is not too hard to fix, but unfortunately I don't have time to fix it for this milestone.

Another thing is the hash map indexing. Since we mod the hash with the array size, we don't have a uniform distribution of the array indices. I don't know a way to get a uniform distribution, so if you know a way please let me know.

Lastly, since my AST is closely related on the structure of the grammar, there were a lot of cases/recursions to type out during the traversal. While it wasn't too hard to follow the cases when I coded it up, the amount of code is pretty large. Again, I think I should have generalized my AST in the previous milestone, but it is too late now.

Identify one specific area in the code you'd like feedback on from the TA (it can be the area you think needs improvement, or a different area). Please identify the filename (and pathname, if applicable) of where that code can be found I don't have exactly one area in mind, but it would be great if you can peek at my `semantics.h` and `semantics.c` code and see what I can do better. By the way, I found the feedback from the previous milestones very helpful, and based on your feedback I changed up my code and it feels better structured and relatively easy to manage than before, so thank you for that :)

[1] `https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function`

What grade you think you've earned on the milestone and why We will give some evidence first:

- *Good error and warning messages output:* please see the test cases below
- *Predictable: correct inputs parsed correctly, erroneous inputs rejected.* we will give some test cases and you can see the error messages yourself (I think they are pretty good, better than the reference compiler's error message for the most part). I will not show the reference compiler's error messages here, but they don't have column numbers, and the information they output is not very specific most of the time (for example, our compiler error tells the user that is has too little or too many arguments when we call the function, and also distinguish which argument has the wrong type, but the reference compiler just says type/number of arguments mismatch).

  The test files can be found at `./testFiles/semanticTests`.

  Here are some test cases:

  ```
  F=../testFiles/semanticTests/toplvTest1.txt ; cat $F; ./semantic.out $F
  int a;

  //this is a main function
  b(){}
  //this is another main function
  c(){}

  void d(){
      //we call non-main functions
      d();
      //but we cannot call a main function
      c();
  }
  ../testFiles/semanticTests/toplvTest1.txt: line 6, col 1: error: more than 1 main is redefined with
  name 'c'
  ../testFiles/semanticTests/toplvTest1.txt: line 12, col 5: error: cannot call the main function 'c'
  ```

  From the above test case, we see that we have multiple declaration of main, and we cannot call the main function. These are reported in the error message.

  ```
  F=../testFiles/semanticTests/toplvTest2.txt ; cat $F; ./semantic.out $F
  //multiple redeclarations of variable names on the top level
  int f;
  boolean f;
  void f(){}

  //according to the reference compiler, only the name matters for declarations
  //so no function "overloading"
  void g(){}

  //there's also no return statement in the 2 functions below
  boolean g(){}
  int g(int f){}

  //no main declaration
  ../testFiles/semanticTests/toplvTest2.txt: line 3, col 9: error: the name 'f' is redefined as a
  variable!
  ../testFiles/semanticTests/toplvTest2.txt: line 4, col 6: error: the name 'f' is redefined as a
  function
  ../testFiles/semanticTests/toplvTest2.txt: line 11, col 9: error: the name 'g' is redefined as a
  function
  ../testFiles/semanticTests/toplvTest2.txt: line 12, col 5: error: the name 'g' is redefined as a
  function
  error: no main function detected!
  ../testFiles/semanticTests/toplvTest2.txt: line 11, col 9: error: no return statement inside a non-void
  function g
  ../testFiles/semanticTests/toplvTest2.txt: line 12, col 5: error: no return statement inside a non-void
  function g
  ```

  We see that `boolean f` and `void f()` are redefinitions of `int f`. Then since only the names matter, we cannot overload any functions (have same name but different return types or arguments), it will just be a redefinition error, as illustrated with `g`. We keep semantic check the body of `g`, and found that they return non-void (boolean or int), but there are no return types corresponding to it.

```
F=../testFiles/semanticTests/redefTest1.txt ; cat $F; ./semantic.out $F
int a;
int b;

//note we have an integer "a" on the top level, and one as an argument to f
void f(int a){
    //this "a" references to the function argument "a"
    a = 1;
    //we can redeclare "b" here
    boolean b;
    //the reference of "b" should be the inner one
    b = false;
}

main(){
    //this "a" references to the global variable "a"
    a = 1;
    {
        //referencing inside blocks are fine
        a = 2;
        {
            {
                a=3;
            }
            {
                a=4;
            }
        }
    }
}
globalVarDecl
  int, {'lineno': 1, 'colno': 1, 'attr': None}
  Identifier, {'lineno': 1, 'colno': 5, 'attr': 'a', 'ref': 0x55ea31e89630}
globalVarDecl
  int, {'lineno': 2, 'colno': 1, 'attr': None}
  Identifier, {'lineno': 2, 'colno': 5, 'attr': 'b', 'ref': 0x55ea31e89790}
funcDecl
    void, {'lineno': 5, 'colno': 1, 'attr': None}
    Identifier, {'lineno': 5, 'colno': 6, 'attr': 'f', 'ref': 0x55ea31e8a190}
      formals:
        int, {'lineno': 5, 'colno': 8, 'attr': None}
        Identifier, {'lineno': 5, 'colno': 12, 'attr': 'a', 'ref': 0x55ea31e899d0}

    block:

      statemExp: {'lineno': 7, 'attr': None}
        Expression: {sig : 'int'}
          = {'lineno': 7, 'colno': 7, 'attr': None}
            Identifier, {'lineno': 7, 'colno': 5, 'attr': 'a', 'ref': 0x55ea31e899d0}
            Expression: {sig : 'int'}
              Number, {'lineno': 7, 'colno': 9, 'attr': '1'}



      varDecl:
        boolean, {'lineno': 9, 'colno': 5, 'attr': None}
        Identifier, {'lineno': 9, 'colno': 13, 'attr': 'b', 'ref': 0x55ea31e89e90}
      statemExp: {'lineno': 11, 'attr': None}
        Expression: {sig : 'boolean'}
          = {'lineno': 11, 'colno': 7, 'attr': None}
            Identifier, {'lineno': 11, 'colno': 5, 'attr': 'b', 'ref': 0x55ea31e89e90}
            Expression: {sig : 'boolean'}
              false, {'lineno': 11, 'colno': 9, 'attr': None}



mainDecl
    void, {'attr': None}
    Identifier, {'lineno': 14, 'colno': 1, 'attr': 'main', 'ref': 0x55ea31e8b270}
      formals:
```

```
    block:

      statemExp: {'lineno': 16, 'attr': None}
        Expression: {sig : 'int'}
          = {'lineno': 16, 'colno': 7, 'attr': None}
            Identifier, {'lineno': 16, 'colno': 5, 'attr': 'a', 'ref': 0x55ea31e89630}
            Expression: {sig : 'int'}
              Number, {'lineno': 16, 'colno': 9, 'attr': '1'}


      block:

        statemExp: {'lineno': 19, 'attr': None}
          Expression: {sig : 'int'}
            = {'lineno': 19, 'colno': 11, 'attr': None}
              Identifier, {'lineno': 19, 'colno': 9, 'attr': 'a', 'ref': 0x55ea31e89630}
              Expression: {sig : 'int'}
                Number, {'lineno': 19, 'colno': 13, 'attr': '2'}


        block:

          block:

            statemExp: {'lineno': 22, 'attr': None}
              Expression: {sig : 'int'}
                = {'lineno': 22, 'colno': 18, 'attr': None}
                  Identifier, {'lineno': 22, 'colno': 17, 'attr': 'a', 'ref': 0x55ea31e89630}
                  Expression: {sig : 'int'}
                    Number, {'lineno': 22, 'colno': 19, 'attr': '3'}


          block:

            statemExp: {'lineno': 25, 'attr': None}
              Expression: {sig : 'int'}
                = {'lineno': 25, 'colno': 18, 'attr': None}
                  Identifier, {'lineno': 25, 'colno': 17, 'attr': 'a', 'ref': 0x55ea31e89630}
                  Expression: {sig : 'int'}
                    Number, {'lineno': 25, 'colno': 19, 'attr': '4'}
```

This is a test of symbol table reference. We declared two global variables `int a` and `int b`. Then inside function `f`, we have an argument also named `a`. But this is fine, since it is on a different scope. Then inside `f`, we refer to `a`, it actually points to the formal parameter address. We also redeclared `b` as a boolean instead of int inside `f`, and that also works with the correct reference pointer. Then inside `main`, we try to refer to the global variable `a` inside blocks, which also works with expected reference pointer.

```
 F=../testFiles/semanticTests/redefTest2.txt ; cat $F; ./semantic.out $F

//redefinition inside function argument
void f(int a, int a){
    //this is a redefinition of "a"!
    //1st instance of "a" is in the function argument
    int a;
    boolean aa;
    int aa; //another redefinition
    {
        //cannot declare local declaration inside an outermost block
        int b;
        {
            int bb; //same here
        }
    }
    if (true){
        int c; //no
    }
    if (true){
        int d; //no
    }
    else{
        int e; //no
    }
    while(false){
        int f; //no
    }
}

main(){
}
../testFiles/semanticTests/redefTest2.txt: line 3, col 19: error: the name 'a' is redefined as a
variable!
../testFiles/semanticTests/redefTest2.txt: line 6, col 9: error: the name 'a' is redefined as a
variable!
../testFiles/semanticTests/redefTest2.txt: line 8, col 9: error: the name 'aa' is redefined as a
variable!
../testFiles/semanticTests/redefTest2.txt: line 11, col 13: error: variable declaration 'b' not in an
outermost block!
../testFiles/semanticTests/redefTest2.txt: line 13, col 17: error: variable declaration 'bb' not in an
outermost block!
../testFiles/semanticTests/redefTest2.txt: line 17, col 13: error: variable declaration 'c' not in an
outermost block!
../testFiles/semanticTests/redefTest2.txt: line 20, col 13: error: variable declaration 'd' not in an
outermost block!
../testFiles/semanticTests/redefTest2.txt: line 23, col 13: error: variable declaration 'e' not in an
outermost block!
../testFiles/semanticTests/redefTest2.txt: line 26, col 13: error: variable declaration 'f' not in an
outermost block!
```

The first error we got is that inside definition of `f`, we redefined the variable `a` inside its arguments. Then we redefined `a` again inside the function body. We declared `aa` as a boolean first then an integer, which also causes a redefinition error. Then we test that we indeed cannot declare local declaration inside an outermost block.

```
F=../testFiles/semanticTests/redefTest3.txt ; cat $F; ./semantic.out $F
int g(){
    //we can decalre the name "g" here, even though "g" is also a function name
    int g;
    //however, when we look up from symbol table, we see that the
    //first occurence of "g" is a variable, not a function
    g(); //this will error
    g=2; //this is fine
    g = h; //cannot treat function "h" as a variable
    return 1;
}
int h() {return 1;}
main(){}
../testFiles/semanticTests/redefTest3.txt: line 6, col 5: error: the variable 'g' is used as a function
../testFiles/semanticTests/redefTest3.txt: line 8, col 9: error: the function 'h' is used as a variable
../testFiles/semanticTests/redefTest3.txt: line 8, col 7: error: in operator '=', the identifier of an
assignment has type 'int', but the expression has type 'UNKNOWN_TYPE'
```

In this short test case, we can redeclare the name `g` as a function and as an integer inside its function body. This will not cause a redefinition error in the reference compiler, and so does in our compiler. However, when we call `g()`, since the most recent `g` in the symbol table is the integer `g`, this will error and say that you are trying to call a function, but it is actually a variable. The symmetric case occurs at `h`, since `h` is defined as a function, when we try to use it as a variable, it will also tell us that. The last error with `UNKNOWN_TYPE` is just a side effect of keep semantic checking with an error, it will propagate up the type as 0 (as a enum), which when printed, translates to `UNKNOWN_TYPE`.

```
F=../testFiles/semanticTests/freeVarTest1.txt ; cat $F; ./semantic.out $F
void b(){}
main(){
    int a;
    //here "a" is a variable but a function invocation "a()" is called
    //also "b" is a function but a variable invocation "b" is called
    a = c + b + a();
    e = 1;  //"e" is not defined yet!
    int e;
    c(a, 1); //undefined function
}
../testFiles/semanticTests/freeVarTest1.txt: line 6, col 9: error: unknown identifier 'c'
../testFiles/semanticTests/freeVarTest1.txt: line 6, col 13: error: the function 'b' is used as a
variable
../testFiles/semanticTests/freeVarTest1.txt: line 6, col 11: error: in binary operator '+', expected
left expression type 'int', but expression has type 'UNKNOWN_TYPE'
../testFiles/semanticTests/freeVarTest1.txt: line 6, col 17: error: the variable 'a' is used as a
function
../testFiles/semanticTests/freeVarTest1.txt: line 6, col 15: error: in binary operator '+', expected
left expression type 'int', but expression has type 'UNKNOWN_TYPE'
../testFiles/semanticTests/freeVarTest1.txt: line 6, col 7: error: in operator '=', the identifier of
an assignment has type 'int', but the expression has type 'UNKNOWN_TYPE'
../testFiles/semanticTests/freeVarTest1.txt: line 7, col 5: error: unknown identifier 'e'
../testFiles/semanticTests/freeVarTest1.txt: line 7, col 5: error: identifier name of an assignment of
type 'UNKNOWN_TYPE' is not a valid type. It should be either a type 'int' or 'boolean'
../testFiles/semanticTests/freeVarTest1.txt: line 9, col 5: error: function name 'c' does not exist
```

This test show case a similar case with the previous test case: if we call a function as a variable or vice versa, it will cause an explicit error. We note that `c` is an free/undefined variable. Same thing with `e`, although it is declared immediately after it is invoked. We also tried to call an unknown function `c`, which is caught and outputs an error.

```
F=../testFiles/semanticTests/functionCallTest1.txt ; cat $F; ./semantic.out $F

void f1(){}
int f2(int a, boolean b){return 1;}
boolean f3(boolean a, int b){return true;}

main(){
    int a;
    a = f1(); //cannot set void to int
    f1(a, true); //too many arguments
    a = f2(true, a); //1st type parameter mismatch
    a = f2(a, a); //2nd type parameter mismatch
    a = f2(a, f3()); //too little parameters for f3
}
../testFiles/semanticTests/functionCallTest1.txt: line 8, col 7: error: in operator '=', the identifier
of an assignment has type 'int', but the expression has type 'void'
../testFiles/semanticTests/functionCallTest1.txt: line 9, col 8: error: too many input arguments to the
function 'f1'
../testFiles/semanticTests/functionCallTest1.txt: line 10, col 12: error: expected type 'int' but got
type 'boolean'
../testFiles/semanticTests/functionCallTest1.txt: line 11, col 15: error: expected type 'boolean' but
got type 'int'
../testFiles/semanticTests/functionCallTest1.txt: line 12, col 15: error: too little input arguments to
the function 'f3'
```

This test case targets the function invocations. First, we note that we cannot set an integer `a` to a void type, and the error message explicitly say that in an assignment, the expression has type void, but we wanted an integer. The next line tries to invoke `f1` with too many arguments. In the next line, the first type of `f2` is an int, but we inputted a bool; the error message shows exactly the column number of the type mismatch, which is great. The same thing occurs on the next line, where the second argument has a mismatch type. The last line is the types match, but the function invocation `f3` has too little input arguments.

```
    F=../testFiles/semanticTests/returnFromFunctionTest1.txt ; cat $F; ./semantic.out $F

void f(){
    //if function is void, we don't need a return statement
}
void g(){
    //but we can add one if it's void
    return;
}
int h(){
    //no return statement, bad
}
```

```
boolean a(){
    //expected return bool, not other types
    int a;
    return a;
    return true;
    return "stringg";
    return; //need to return a value
}
void b(){
    //expected to not return any value, but we did
    return 1;
    if (true)
        //note in languages like C, return void is allowed since void is not a value?
        //but the reference compiler doesn't allow that, so we will also error here
        return b();
    else
        return h();
}
main(){}
../testFiles/semanticTests/returnFromFunctionTest1.txt: line 9, col 5: error: no return statement
inside a non-void function h
../testFiles/semanticTests/returnFromFunctionTest1.txt: line 15, col 5: error: in return statement,
expected return type of 'boolean' but got 'int'
../testFiles/semanticTests/returnFromFunctionTest1.txt: line 17, col 5: error: in return statement,
expected return type of 'boolean' but got 'string'
../testFiles/semanticTests/returnFromFunctionTest1.txt: line 18, col 5: error: this function returns
non-void but it doesn't have a return a value
../testFiles/semanticTests/returnFromFunctionTest1.txt: line 22, col 12: error: this function returns
void and thus cannot return a value
../testFiles/semanticTests/returnFromFunctionTest1.txt: line 26, col 0: error: this function returns
void and thus cannot return a value
../testFiles/semanticTests/returnFromFunctionTest1.txt: line 28, col 0: error: this function returns
void and thus cannot return a value
```

We move on to function return statements. In `f`, the function is void so we don't need a return statement; but we can add one if we want, which is done inside `g`. In `h`, we don't have a return statement in a non-void function, and the error message tells us exactly that. In `a`, we try to return many different types, but on line 15, 17, 18, we returned a different type than boolean, which we also have an explicit expected vs actual return type error. In `b`, we cannot return any type (including the void type) inside a void function.

```
F=../testFiles/semanticTests/statementExpTest1.txt ; cat $F; ./semantic.out $F
main(){
    int a;
    //undefined variable error, and statement exp error
    b+1;
    //below are some more statement exp errors
    -1;
    !true;
    3+1;
    b = 1;
}
../testFiles/semanticTests/statementExpTest1.txt: line 4, col 6: error: must be assignment or function
call inside a statement expression
../testFiles/semanticTests/statementExpTest1.txt: line 4, col 5: error: must be assignment or function
call inside a statement expression
../testFiles/semanticTests/statementExpTest1.txt: line 4, col 5: error: unknown identifier 'b'
../testFiles/semanticTests/statementExpTest1.txt: line 4, col 7: error: must be assignment or function
call inside a statement expression
../testFiles/semanticTests/statementExpTest1.txt: line 4, col 6: error: in binary operator '+',
expected left expression type 'int', but expression has type 'UNKNOWN_TYPE'
../testFiles/semanticTests/statementExpTest1.txt: line 6, col 5: error: must be assignment or function
call inside a statement expression
../testFiles/semanticTests/statementExpTest1.txt: line 6, col 6: error: must be assignment or function
call inside a statement expression
../testFiles/semanticTests/statementExpTest1.txt: line 7, col 5: error: must be assignment or function
call inside a statement expression
../testFiles/semanticTests/statementExpTest1.txt: line 7, col 6: error: must be assignment or function
call inside a statement expression
```

```
../testFiles/semanticTests/statementExpTest1.txt: line 8, col 6: error: must be assignment or function
call inside a statement expression
../testFiles/semanticTests/statementExpTest1.txt: line 8, col 5: error: must be assignment or function
call inside a statement expression
../testFiles/semanticTests/statementExpTest1.txt: line 8, col 7: error: must be assignment or function
call inside a statement expression
../testFiles/semanticTests/statementExpTest1.txt: line 9, col 5: error: unknown identifier 'b'
../testFiles/semanticTests/statementExpTest1.txt: line 9, col 5: error: identifier name of an
assignment of type 'UNKNOWN_TYPE' is not a valid type. It should be either a type 'int' or 'boolean'
```

This test case is targeted towards statement expressions. The way the semantic check is implemented is that if we are inside a statement expression, a flag is set and passed down, so that whenever we encounter a subexpression that is not an assignment nor a function call, it will error saying that such expression should not exist. Since we keep checking the expression body when we encounter an error, there are many of such errors. But the point is that we correctly detects if an expression is inside a statement expression and we will error accordingly.

```
F=../testFiles/semanticTests/mainArgs.txt ; cat $F; ./semantic.out $F
main(int a){}//this should be a parse error
../testFiles/semanticTests/mainArgs.txt: line 1, col 6: syntax error:  expected ) before int
```

This small test case shows that we covered the case where main declaration can't have arguments. This was actually covered during parsing, so the error message is not great.

```
F=../testFiles/semanticTests/typeMatchTest1.txt ; cat $F; ./semantic.out $F
int f(){ return 1;}
void g(){}
main(){
    boolean b;
    b= 2<true;
    b= 2<=3>4; //left side should be bool instead of int
    b= 2>=3||4; //right side should be int
    b= 2<3&&f(); //ok
    b= 2<3 != g(); //right side is void, bad
    b= f() == 1; //ok
    b= g() == g(); //cannot compare type void
    b= "a">= 1; //bad
    b= 3  > 1; //ok
    b= !true; //ok
    b= !1; //bad

}
../testFiles/semanticTests/typeMatchTest1.txt: line 5, col 9: error: in binary operator '<', expected
right expression type 'int', but expression has type 'boolean'
../testFiles/semanticTests/typeMatchTest1.txt: line 6, col 12: error: in binary operator '>', expected
left expression type 'int', but expression has type 'boolean'
../testFiles/semanticTests/typeMatchTest1.txt: line 6, col 6: error: in operator '=', the identifier of
an assignment has type 'boolean', but the expression has type 'UNKNOWN_TYPE'
../testFiles/semanticTests/typeMatchTest1.txt: line 7, col 12: error: in binary operator '||', expected
right expression type 'boolean', but expression has type 'int'
../testFiles/semanticTests/typeMatchTest1.txt: line 8, col 11: error: in binary operator '&&', expected
right expression type 'boolean', but expression has type 'int'
../testFiles/semanticTests/typeMatchTest1.txt: line 9, col 12: error: in binary operator '!=', expected
right expression type 'boolean', but expression has type 'void'
../testFiles/semanticTests/typeMatchTest1.txt: line 11, col 12: error: in binary operator '==',
expected left expression type to be either 'int' or 'boolean', but got type 'void'
../testFiles/semanticTests/typeMatchTest1.txt: line 11, col 6: error: in operator '=', the identifier
of an assignment has type 'boolean', but the expression has type 'UNKNOWN_TYPE'
../testFiles/semanticTests/typeMatchTest1.txt: line 12, col 11: error: in binary operator '>=',
expected left expression type 'int', but expression has type 'string'
../testFiles/semanticTests/typeMatchTest1.txt: line 12, col 6: error: in operator '=', the identifier
of an assignment has type 'boolean', but the expression has type 'UNKNOWN_TYPE'
../testFiles/semanticTests/typeMatchTest1.txt: line 15, col 8: error: in unary operator '!', expected
expression type 'boolean', but expression has type 'int'
../testFiles/semanticTests/typeMatchTest1.txt: line 15, col 6: error: in operator '=', the identifier
of an assignment has type 'boolean', but the expression has type 'UNKNOWN_TYPE'
```

This test case focus on the types matching for the boolean return type operators. The comments inside the file gives a pretty good indication of what is expected, and the error messages matches what we expect vs the actual type used. We also note that in most cases, the left or the right direction is also indicated in the error message, so that it is easier to see which side of the binary expression is wrong.

Again we have some `UNKNOWN_TYPE`, but that is just a side effect of keep error checking an invalid program. We could easily fix these up to return the expected type so that the user sees less error messages, but I think can be a good indication to the programmer where exactly went wrong.

```
F=../testFiles/semanticTests/typeMatchTest2.txt ; cat $F; ./semantic.out $F
int f(){ return 1;}
void g(){}
main(){
    int b;
    b= 2<true; //2 errors here
    b= 3- f(); //ok
    b= 2 + g(); //right side is void, bad
    b= f() * 1; //ok
    b= "a" / 1; //bad
    b= 3  % 1; //ok
    b= -2; //ok

}
../testFiles/semanticTests/typeMatchTest2.txt: line 5, col 9: error: in binary operator '<', expected
right expression type 'int', but expression has type 'boolean'
../testFiles/semanticTests/typeMatchTest2.txt: line 5, col 6: error: in operator '=', the identifier of
an assignment has type 'int', but the expression has type 'boolean'
../testFiles/semanticTests/typeMatchTest2.txt: line 7, col 10: error: in binary operator '+', expected
right expression type 'int', but expression has type 'void'
../testFiles/semanticTests/typeMatchTest2.txt: line 9, col 12: error: in binary operator '/', expected
left expression type 'int', but expression has type 'string'
../testFiles/semanticTests/typeMatchTest2.txt: line 9, col 6: error: in operator '=', the identifier of
an assignment has type 'int', but the expression has type 'UNKNOWN_TYPE'
```

This test case focus on the types matching for integer return type operators. The comments in the file and error messages gives a good indication on where went wrong.

```
F=../testFiles/semanticTests/ifTest1.txt ; cat $F; ./semantic.out $F
void f(){}

main(){
    if (true); //this is ok
    //should error since not a bool
    if (1);
    if ("string");
    if (f());

    //if-else statement is treated the same way
    //the error message doesn't distinguish whether if the error comes from
    //if only block, or an if else block. But it can be easily done to do so
    if (true); else; //this is ok
    //should error since not a bool
    if (1); else;
    if ("string"); else;
    if (f()); else;
}
../testFiles/semanticTests/ifTest1.txt: line 6, col 5: error: need a boolean expression inside an if
statement condition
../testFiles/semanticTests/ifTest1.txt: line 7, col 5: error: need a boolean expression inside an if
statement condition
../testFiles/semanticTests/ifTest1.txt: line 8, col 5: error: need a boolean expression inside an if
statement condition
../testFiles/semanticTests/ifTest1.txt: line 15, col 5: error: need a boolean expression inside an if
statement condition
../testFiles/semanticTests/ifTest1.txt: line 16, col 5: error: need a boolean expression inside an if
statement condition
../testFiles/semanticTests/ifTest1.txt: line 17, col 5: error: need a boolean expression inside an if
statement condition
```

This tests the if statements. Specifically, we check that the if statement condition must be a boolean. The if else statement is treated the same way, and the error message doesn't change. We could easily output which type of if statement (only if, or if else) in the error message, but I don't think it is necessary. The column number already indicates the location of the if clause, so that should be enough.

```
F=../testFiles/semanticTests/whileTest1.txt ; cat $F; ./semantic.out $F
void f(){}
main(){
    //cannot break when not inside a while loop
    break;
    {
        { break;} //bad
    }
    if (true) break; //bad
    if (true) {break;} //bad
    while(true){
        break; //ok
        {
            break;//ok
        }
        while(false) break; //also ok
    }

    //these should fail
    while (1);
    while ("string");
    while (f());
}
../testFiles/semanticTests/whileTest1.txt: line 4, col 5: error: break must be inside a while loop
../testFiles/semanticTests/whileTest1.txt: line 6, col 11: error: break must be inside a while loop
../testFiles/semanticTests/whileTest1.txt: line 8, col 15: error: break must be inside a while loop
../testFiles/semanticTests/whileTest1.txt: line 9, col 16: error: break must be inside a while loop
../testFiles/semanticTests/whileTest1.txt: line 19, col 5: error: need a boolean expression inside a
while loop condition
../testFiles/semanticTests/whileTest1.txt: line 20, col 5: error: need a boolean expression inside a
while loop condition
../testFiles/semanticTests/whileTest1.txt: line 21, col 5: error: need a boolean expression inside a
while loop condition
```

This tests the while statements. It tests that we can only break inside a while loop (break inside blocks and if statements are tested). We also test that the while loop condition must be a boolean type.

```
F=../testFiles/semanticTests/builtinTypesTest1.txt ; cat $F; ./semantic.out $F
//pre-defined library functions:
//
//int getchar()
//void halt()
//void printb(boolean b)
//void printc(int c)
//void printi(int i)
//void prints(string s)

main(){
    int a;
    boolean b;
    a = getchar();
    halt();
    printb(b);
    printc(a);
    printi(1);
    prints("stringg");
}

mainDecl
    void, {'attr': None}
    Identifier, {'lineno': 10, 'colno': 1, 'attr': 'main', 'ref': 0x55d41bd98c80}
      formals:

    block:
```

```
    varDecl:
      int, {'lineno': 11, 'colno': 5, 'attr': None}
      Identifier, {'lineno': 11, 'colno': 9, 'attr': 'a', 'ref': 0x55d41bd97790}
    varDecl:
      boolean, {'lineno': 12, 'colno': 5, 'attr': None}
      Identifier, {'lineno': 12, 'colno': 13, 'attr': 'b', 'ref': 0x55d41bd978f0}
    statemExp: {'lineno': 13, 'attr': None}
      Expression: {sig : 'int'}
        = {'lineno': 13, 'colno': 7, 'attr': None}
          Identifier, {'lineno': 13, 'colno': 5, 'attr': 'a', 'ref': 0x55d41bd97790}
          Expression: {sig : 'int'}
            funcCall{'lineno': 13, 'attr': None, 'ref': 0x55d41bd9a540}
              Identifier, {'lineno': 13, 'colno': 9, 'attr': 'getchar'}
              func call args:




    statemExp: {'lineno': 14, 'attr': None}
      Expression: {sig : 'void'}
        funcCall{'lineno': 14, 'attr': None, 'ref': 0x55d41bd9a6a0}
          Identifier, {'lineno': 14, 'colno': 5, 'attr': 'halt'}
          func call args:



    statemExp: {'lineno': 15, 'attr': None}
      Expression: {sig : 'void'}
        funcCall{'lineno': 15, 'attr': None, 'ref': 0x55d41bd9a920}
          Identifier, {'lineno': 15, 'colno': 5, 'attr': 'printb'}
          func call args:
            Expression: {sig : 'boolean'}
              Identifier, {'lineno': 15, 'colno': 12, 'attr': 'b', 'ref': 0x55d41bd978f0}



    statemExp: {'lineno': 16, 'attr': None}
      Expression: {sig : 'void'}
        funcCall{'lineno': 16, 'attr': None, 'ref': 0x55d41bd9aba0}
          Identifier, {'lineno': 16, 'colno': 5, 'attr': 'printc'}
          func call args:
            Expression: {sig : 'int'}
              Identifier, {'lineno': 16, 'colno': 12, 'attr': 'a', 'ref': 0x55d41bd97790}



    statemExp: {'lineno': 17, 'attr': None}
      Expression: {sig : 'void'}
        funcCall{'lineno': 17, 'attr': None, 'ref': 0x55d41bd9ae20}
          Identifier, {'lineno': 17, 'colno': 5, 'attr': 'printi'}
          func call args:
            Expression: {sig : 'int'}
              Number, {'lineno': 17, 'colno': 12, 'attr': '1'}



    statemExp: {'lineno': 18, 'attr': None}
      Expression: {sig : 'void'}
        funcCall{'lineno': 18, 'attr': None, 'ref': 0x55d41bd9b0a0}
          Identifier, {'lineno': 18, 'colno': 5, 'attr': 'prints'}
          func call args:
            Expression: {sig : 'string'}
              string, {'lineno': 18, 'colno': 20, 'attr': '"stringg"'}
```

This test case focuses on the builtin functions. Note that it checks successfully, and we have symbol table pointers to them (although we didn't print their corresponding address out, inside the code we pre-define them so the pointer addresses should be fine).

```
F=../testFiles/semanticTests/builtinTypesTest2.txt ; cat $F; ./semantic.out $F
//we can also override a builtin function!
int printi(boolean a, int b){ return b;}
main(){
    boolean hello;
    int world;
    world = printi(hello, world);
}
funcDecl
    int, {'lineno': 2, 'colno': 1, 'attr': None}
    Identifier, {'lineno': 2, 'colno': 5, 'attr': 'printi', 'ref': 0x5642df55cb40}
      formals:
        boolean, {'lineno': 2, 'colno': 12, 'attr': None}
        Identifier, {'lineno': 2, 'colno': 20, 'attr': 'a', 'ref': 0x5642df55c710}
        int, {'lineno': 2, 'colno': 23, 'attr': None}
        Identifier, {'lineno': 2, 'colno': 27, 'attr': 'b', 'ref': 0x5642df55c870}

    block:

      return: {'lineno': 2, 'colno': 31, 'attr': None}
        Expression: {sig : 'int'}
          Identifier, {'lineno': 2, 'colno': 38, 'attr': 'b', 'ref': 0x5642df55c870}

mainDecl
    void, {'attr': None}
    Identifier, {'lineno': 3, 'colno': 1, 'attr': 'main', 'ref': 0x5642df55d600}
      formals:

    block:

      varDecl:
        boolean, {'lineno': 4, 'colno': 5, 'attr': None}
        Identifier, {'lineno': 4, 'colno': 13, 'attr': 'hello', 'ref': 0x5642df55ce00}
      varDecl:
        int, {'lineno': 5, 'colno': 5, 'attr': None}
        Identifier, {'lineno': 5, 'colno': 9, 'attr': 'world', 'ref': 0x5642df55cf60}
      statemExp: {'lineno': 6, 'attr': None}
        Expression: {sig : 'int'}
          = {'lineno': 6, 'colno': 11, 'attr': None}
            Identifier, {'lineno': 6, 'colno': 5, 'attr': 'world', 'ref': 0x5642df55cf60}
            Expression: {sig : 'int'}
              funcCall{'lineno': 6, 'attr': None, 'ref': 0x5642df55cb40}
                Identifier, {'lineno': 6, 'colno': 13, 'attr': 'printi'}
                func call args:
                  Expression: {sig : 'boolean'}
                    Identifier, {'lineno': 6, 'colno': 20, 'attr': 'hello', 'ref': 0x5642df55ce00}
                  Expression: {sig : 'int'}
                    Identifier, {'lineno': 6, 'colno': 27, 'attr': 'world', 'ref': 0x5642df55cf60}
```

This test case shows we can overload the builtin functions (although we only overloaded one function here). Since the builtin functions are on the bottom of the stack, when we create the top level declarations we can redeclare them and then call it inside other functions. Since the top level declaration is before the builtin declarations, we can safely call the overload builtin functions. Here we overloaded `printi` by changing its return type and input arguments. Then when we call it inside `main`, we see that the reference pointer points to the `printi` declared inside the program.

```
F=../testFiles/semanticTests/referenceTest1.txt ; cat $F; ./semantic.out $F
int a; //global variable "a"
void f(int a){
    a =1; //should refer to the argument "a"
}
int g(int c, boolean b){
    a=1;//should refer to global variable "a"
    int a;
    a=2;//should refer to local variable "a"
    while (c<=2){
        if (b){
            c = 1;
        }
        else{
            return c;
        }
    }
    return a;
}

boolean h(int d1, boolean d2, int d3, boolean d4,int d5){
    return d1<=d3 || d2==true != d4 && d5<3;
}

main(){
    int d1; boolean d2; int d3; boolean d4; int d5;
    if (g(a, 3<=a) != d3){
        h(d5, d4, d3, d2, d1);
    }
}
globalVarDecl
  int, {'lineno': 1, 'colno': 1, 'attr': None}
  Identifier, {'lineno': 1, 'colno': 5, 'attr': 'a', 'ref': 0x55896f98e630}
funcDecl
    void, {'lineno': 2, 'colno': 1, 'attr': None}
    Identifier, {'lineno': 2, 'colno': 6, 'attr': 'f', 'ref': 0x55896f98ec50}
      formals:
        int, {'lineno': 2, 'colno': 8, 'attr': None}
        Identifier, {'lineno': 2, 'colno': 12, 'attr': 'a', 'ref': 0x55896f98e870}

    block:

      statemExp: {'lineno': 3, 'attr': None}
        Expression: {sig : 'int'}
          = {'lineno': 3, 'colno': 7, 'attr': None}
            Identifier, {'lineno': 3, 'colno': 5, 'attr': 'a', 'ref': 0x55896f98e870}
            Expression: {sig : 'int'}
              Number, {'lineno': 3, 'colno': 8, 'attr': '1'}


funcDecl
    int, {'lineno': 5, 'colno': 1, 'attr': None}
    Identifier, {'lineno': 5, 'colno': 5, 'attr': 'g', 'ref': 0x55896f9905d0}
      formals:
        int, {'lineno': 5, 'colno': 7, 'attr': None}
        Identifier, {'lineno': 5, 'colno': 11, 'attr': 'c', 'ref': 0x55896f98ee90}
        boolean, {'lineno': 5, 'colno': 14, 'attr': None}
        Identifier, {'lineno': 5, 'colno': 22, 'attr': 'b', 'ref': 0x55896f98eff0}

    block:

      statemExp: {'lineno': 6, 'attr': None}
        Expression: {sig : 'int'}
          = {'lineno': 6, 'colno': 6, 'attr': None}
            Identifier, {'lineno': 6, 'colno': 5, 'attr': 'a', 'ref': 0x55896f98e630}
            Expression: {sig : 'int'}
              Number, {'lineno': 6, 'colno': 7, 'attr': '1'}


      varDecl:
        int, {'lineno': 7, 'colno': 5, 'attr': None}
        Identifier, {'lineno': 7, 'colno': 9, 'attr': 'a', 'ref': 0x55896f98f4b0}
      statemExp: {'lineno': 8, 'attr': None}
        Expression: {sig : 'int'}
          = {'lineno': 8, 'colno': 6, 'attr': None}
            Identifier, {'lineno': 8, 'colno': 5, 'attr': 'a', 'ref': 0x55896f98f4b0}
            Expression: {sig : 'int'}
              Number, {'lineno': 8, 'colno': 7, 'attr': '2'}


      while: {'lineno': 9, 'colno': 5, 'attr': None}
        Expression: {sig : 'boolean'}
          <= {'lineno': 9, 'colno': 13, 'attr': None}
            Expression: {sig : 'int'}
              Identifier, {'lineno': 9, 'colno': 12, 'attr': 'c', 'ref': 0x55896f98ee90}
            Expression: {sig : 'int'}
```

```
              Number, {'lineno': 9, 'colno': 15, 'attr': '2'}

          block:

            ifElse {'lineno': 10, 'colno': 9, 'attr': None}
              Expression: {sig : 'boolean'}
                Identifier, {'lineno': 10, 'colno': 13, 'attr': 'b', 'ref': 0x55896f98eff0}
              block:

                statemExp: {'lineno': 11, 'attr': None}
                  Expression: {sig : 'int'}
                    = {'lineno': 11, 'colno': 15, 'attr': None}
                      Identifier, {'lineno': 11, 'colno': 13, 'attr': 'c', 'ref': 0x55896f98ee90}
                      Expression: {sig : 'int'}
                        Number, {'lineno': 11, 'colno': 17, 'attr': '1'}


              block:

                return: {'lineno': 14, 'colno': 13, 'attr': None}
                  Expression: {sig : 'int'}
                    Identifier, {'lineno': 14, 'colno': 20, 'attr': 'c', 'ref': 0x55896f98ee90}

        return: {'lineno': 17, 'colno': 5, 'attr': None}
          Expression: {sig : 'int'}
            Identifier, {'lineno': 17, 'colno': 12, 'attr': 'a', 'ref': 0x55896f98f4b0}

funcDecl
    boolean, {'lineno': 20, 'colno': 1, 'attr': None}
    Identifier, {'lineno': 20, 'colno': 9, 'attr': 'h', 'ref': 0x55896f991840}
      formals:
        int, {'lineno': 20, 'colno': 11, 'attr': None}
        Identifier, {'lineno': 20, 'colno': 15, 'attr': 'd1', 'ref': 0x55896f990810}
        boolean, {'lineno': 20, 'colno': 19, 'attr': None}
        Identifier, {'lineno': 20, 'colno': 27, 'attr': 'd2', 'ref': 0x55896f990970}
        int, {'lineno': 20, 'colno': 31, 'attr': None}
        Identifier, {'lineno': 20, 'colno': 35, 'attr': 'd3', 'ref': 0x55896f990ad0}
        boolean, {'lineno': 20, 'colno': 39, 'attr': None}
        Identifier, {'lineno': 20, 'colno': 47, 'attr': 'd4', 'ref': 0x55896f990c30}
        int, {'lineno': 20, 'colno': 50, 'attr': None}
        Identifier, {'lineno': 20, 'colno': 54, 'attr': 'd5', 'ref': 0x55896f990d90}

    block:

      return: {'lineno': 21, 'colno': 5, 'attr': None}
        Expression: {sig : 'boolean'}
          || {'lineno': 21, 'colno': 19, 'attr': None}
            Expression: {sig : 'boolean'}
              <= {'lineno': 21, 'colno': 14, 'attr': None}
                Expression: {sig : 'int'}
                  Identifier, {'lineno': 21, 'colno': 12, 'attr': 'd1', 'ref': 0x55896f990810}
                Expression: {sig : 'int'}
                  Identifier, {'lineno': 21, 'colno': 16, 'attr': 'd3', 'ref': 0x55896f990ad0}

            Expression: {sig : 'boolean'}
              && {'lineno': 21, 'colno': 37, 'attr': None}
                Expression: {sig : 'boolean'}
                  != {'lineno': 21, 'colno': 31, 'attr': None}
                    Expression: {sig : 'boolean'}
                      == {'lineno': 21, 'colno': 24, 'attr': None}
                        Expression: {sig : 'boolean'}
                          Identifier, {'lineno': 21, 'colno': 22, 'attr': 'd2', 'ref': 0x55896f990970}
                        Expression: {sig : 'boolean'}
                          true, {'lineno': 21, 'colno': 26, 'attr': None}

                    Expression: {sig : 'boolean'}
                      Identifier, {'lineno': 21, 'colno': 34, 'attr': 'd4', 'ref': 0x55896f990c30}
            Expression: {sig : 'boolean'}
              < {'lineno': 21, 'colno': 42, 'attr': None}
                Expression: {sig : 'int'}
                  Identifier, {'lineno': 21, 'colno': 40, 'attr': 'd5', 'ref': 0x55896f990d90}
                Expression: {sig : 'int'}
                  Number, {'lineno': 21, 'colno': 43, 'attr': '3'}




mainDecl
    void, {'attr': None}
    Identifier, {'lineno': 24, 'colno': 1, 'attr': 'main', 'ref': 0x55896f993210}
      formals:

    block:

      varDecl:
        int, {'lineno': 25, 'colno': 5, 'attr': None}
        Identifier, {'lineno': 25, 'colno': 9, 'attr': 'd1', 'ref': 0x55896f991b00}
      varDecl:
        boolean, {'lineno': 25, 'colno': 13, 'attr': None}
        Identifier, {'lineno': 25, 'colno': 21, 'attr': 'd2', 'ref': 0x55896f991c60}
      varDecl:
        int, {'lineno': 25, 'colno': 25, 'attr': None}
        Identifier, {'lineno': 25, 'colno': 29, 'attr': 'd3', 'ref': 0x55896f991dc0}
      varDecl:
        boolean, {'lineno': 25, 'colno': 33, 'attr': None}
        Identifier, {'lineno': 25, 'colno': 41, 'attr': 'd4', 'ref': 0x55896f991f20}
      varDecl:
        int, {'lineno': 25, 'colno': 45, 'attr': None}
        Identifier, {'lineno': 25, 'colno': 49, 'attr': 'd5', 'ref': 0x55896f992080}
      if {'lineno': 26, 'colno': 5, 'attr': None}
        Expression: {sig : 'boolean'}
          != {'lineno': 26, 'colno': 20, 'attr': None}
            Expression: {sig : 'int'}
              funcCall{'lineno': 26, 'attr': None, 'ref': 0x55896f9905d0}
                Identifier, {'lineno': 26, 'colno': 9, 'attr': 'g'}
                func call args:
                  Expression: {sig : 'int'}
                    Identifier, {'lineno': 26, 'colno': 11, 'attr': 'a', 'ref': 0x55896f98e630}
                  Expression: {sig : 'boolean'}
                    <= {'lineno': 26, 'colno': 15, 'attr': None}
                      Expression: {sig : 'int'}
                        Number, {'lineno': 26, 'colno': 14, 'attr': '3'}
                      Expression: {sig : 'int'}
                        Identifier, {'lineno': 26, 'colno': 17, 'attr': 'a', 'ref': 0x55896f98e630}


            Expression: {sig : 'int'}
              Identifier, {'lineno': 26, 'colno': 23, 'attr': 'd3', 'ref': 0x55896f991dc0}

        block:

          statemExp: {'lineno': 27, 'attr': None}
            Expression: {sig : 'boolean'}
              funcCall{'lineno': 27, 'attr': None, 'ref': 0x55896f991840}
                Identifier, {'lineno': 27, 'colno': 9, 'attr': 'h'}
                func call args:
                  Expression: {sig : 'int'}
                    Identifier, {'lineno': 27, 'colno': 11, 'attr': 'd5', 'ref': 0x55896f992080}
                  Expression: {sig : 'boolean'}
                    Identifier, {'lineno': 27, 'colno': 15, 'attr': 'd4', 'ref': 0x55896f991f20}
                  Expression: {sig : 'int'}
                    Identifier, {'lineno': 27, 'colno': 19, 'attr': 'd3', 'ref': 0x55896f991dc0}
                  Expression: {sig : 'boolean'}
                    Identifier, {'lineno': 27, 'colno': 23, 'attr': 'd2', 'ref': 0x55896f991c60}
                  Expression: {sig : 'int'}
```

```
Identifier, {'lineno': 27, 'colno': 27, 'attr': 'd1', 'ref': 0x55896f991b00}
```

This last test case demonstrates the symbol table reference should be correct. The details of chasing through if the reference refers to the correct variable/function call is left to the reader, but I personally checked it and it looks fine.

- *Committing work regularly to your CPSC Gitlab repo:* I have a reasonable amount of commits, and I push when I get a big chunk done so I don't lose the files accidentally. I admit that I didn't complete the assignment very early on, but I had assignments and thesis work due near the end of the semester, and I felt sick for the past two weeks, so I couldn't get much done very early on.
- *Code is modular:* I think my code is reasonably modular. I put my hashmap and stack code inside `hashtable.h` and `hashtable.c`, and some useful functions separate from the semantic checking in there. Although I did not implement a generic hashmap (because we will need to do more pointer casting), I think except the type of the hashnode we store, everything else seems quite generic. My semantic checking files are inside `semantics.h` and `semantics.c`, which also includes helpful helper functions (such as set and unset the MSB). Inside `semantics.c`, I tried to reduce duplication of code by writing more general functions like `addVarDeclToHashtable` and `checksTypeOfTwoOperators`, which reduced some code duplication overall, especially for type checking the unary and binary operators.
- *Code is extensible:* I think my code is reasonably extensible. For default initializing the structures, there are specific functions for that, and some default values (that might change in the future) are put inside macros. I pass structures up and down, which means if we want to add something when we recurs down, we can easily do that. For symbol table pointers (that points to an AST node), I point to the structure that gives the most possible information, instead of just pointing to the variable name or function header.

  One might argue that the two traversals are not extensible, and I would also agree that it is not, in the sense that we cannot reuse the traversal code again. However, I don't think doing a generic traversal here is really necessary, and it is not too hard to rewrite the traversal for a different task. Traversing many times increases the time complexity linearly, but it really feels overkill to do a generic traversal with many passes, verses I can just have enough information to do it in 2 passes. As I mentioned above, the 1st traversal is to declare the global variables (and see if there are any name clashes), and the second traversal is to semantically check everything else. It also feels natural to do it this way, rather than traverse many times to do the things separately.

  If we really wants generic code, I think the general recursion schemes are great, but it is very hard to implement in C, and not mentioning that I am not good at recursion schemes either.
- *Code is consistent e.g., with respect to variable and function naming conventions:* I think my naming conventions are mostly consistent. For example, the global variables and enum members are all in capital letters with underscores. The structure and variable names are camel cased. If you want to semantic check some struct, the functions have the structure name as a prefix and `SemanticCheck` as a suffix. For example, the functions corresponding to the structure `Expression` is `expressionSemanticCheck`.
- *Code is consistent with respect to the implementation language, i.e., are appropriate idioms and libraries used?:* I think the code is consistent with the implementation language. It is just plain C, with the addition of hashmap and a stack, which is implemented using structures and arrays. To store the pointer value inside the hashmap, I used `uintptr_t`, which is an integer type that, if it is a valid pointer address, can be freely casted to `void*` and back without any issues (well defined behaviour). Since sometimes I need to set the MSB when I store the pointer values, storing the pointer as `uintptr_t` inside the hash table is appropriate.
- *Code is well documented:* I have comments almost everywhere that explains what I'm doing. This way, when I'm writing the code it can keep me on track of what I need to do, and it is also beneficial for the reader to know what I'm doing.
- *Speed: code should not be unreasonably slow* I think my code is pretty fast. For a reference, if we time the reference compiler to semantically check the file (also count the printing time) `testFiles/semanticTests/referenceTest1.txt`, we have the timing

  ```
  real 0m0.045s
  user 0m0.034s
  sys     0m0.009s
  ```

  verses my compiler has timing

  ```
  real 0m0.003s
  user 0m0.001s
  sys     0m0.002s
  ```

  so it is about 10 times faster than the reference compiler.
- *Easy to build, and building without errors or warnings:* Here is the following build output on the CPSC machines:

```
haiyang.he@csx:~/cpsc411/src$ make semantic
flex lexer.l
bison -v -Wall -d parser.y
parser.y: warning: 1 shift/reduce conflict [-Wconflicts-sr]
parser.y: note: rerun with option '-Wcounterexamples' to generate conflict counterexamples
gcc -Wall -g -c semantics.c
gcc -Wall -g semanticMain.c scanner.o commonHeader.o parser.tab.o ast.o hashtable.o semantics.o -lfl -o sem
```

so it builds fine with 1 shift reduce conflict. If we compile with `-Wcounterexamples` we can see the conflict is the dangling else. I cannot show it here since latex doesn't compile with a unicode symbol inside the verbatim environment.

- *Tool usage, if applicable:* see milestone 2, but I used bison and flex. For this milestone no new tools are added: we are just semantically checking the AST generated from the parser.

- *Consistent with the environment: error and warning messages sent to stderr, regular output sent to stdout, exit status code set appropriately:* I print the warning and error messages with stderr, and otherwise we print to stdout. I also set the exit code correctly in `./semanticMain.c` by returning `exit(EXIT_FAILURE)` if either the parser or the scanner had an error, or if there exists a warning, and `return 0` otherwise, which indicates a successful execution.

  There is a macro `MAX_WARNINGS` inside `commonHeader.h`, which is set to 20, and inside my logging functions, it increments the `TOTAL_WARNINGS` global variable. When it reaches higher than `MAX_WARNINGS` we will also exit failure.

- *Easy to run:* I did what the specification said, so I only allowed 1 input argument for my semantic checker, so I think it is easy to run.

- *Meta-documentation: is the way to build and run your milestone code clearly documented in the repo's README.md ?* Yes you can find the `README.md` which explains how to build and run the milestone.

Finally, I think I did what the milestone should do. I tested all of the test cases with the reference compiler, and we have the same expected behaviour. My compiler has better error messages and runs faster than the reference compiler, which I think that is nice to have. I also have put lots of thought to make this work like how it did (it is not just some hacky solution).

Thus overall I think I should get somewhere between 7/8 and 8/8, but I will let you decide on this.