```
joshua@192:~/Documents/Code/splang/midreport/tests (master)$ cat tests.log
*** Compiling ./tests/fail_ambi.spl ***
// Ambiguous input, and typing error
Void main(){
        if(5)
        if(True)
        print(1);
        else
        print(2);
}
*** Done reading file
*** Lexing is done
./tests/fail_ambi.spl:0:0: error: Ambiguous input - able to derive multiple programs
./tests/fail_ambi.spl:0:0: note: Possible interpretation:
Void main(){
        if(5)
                if(True)
                        print(1);
        else
                print(2);
}
./tests/fail_ambi.spl:0:0: note: Possible interpretation:
Void main(){
        if(5)
                if(True)
                        print(1);
                else
                        print(2);
}
*** Parsing failed (but we try to continue!)
*** Scoping succeeded
./tests/fail_ambi.spl:3:5: error: Cannot unify types Int and Bool
        if(5)
           ^
./tests/fail_ambi.spl:3:5: note: Type Int inferred here:
        if(5)
           ^
./tests/fail_ambi.spl:3:5: note: Type Bool inferred here:
        if(5)
           ^
*** Typing failed (but we try to continue!)
// main :: Void()
Void main/*6*/(){
        if(5)
                if(True)
                        print/*0*/(1);
        else
                print/*0*/(2);
}
*** Done


*** Compiling ./tests/fail_arguments.spl ***
// Tim  Steenvoorden (s0712663)
// extended by also including typing errors (so we should see more than one error)

[Int] function (Int n, Bool b, [Int] l)
{
        print(n);
        print(b);
        print(l);
        return l;
}

Void main ()
{
        [Int] list = 1:2:3:[];

        function(10, 10);
        function(10, [], [], []);
}


*** Done reading file
*** Lexing is done
*** Parsing succeeded
*** Scoping succeeded
./tests/fail_arguments.spl:16:2: error: Too few arguments given, 2 given, but 3 expected.
        function(10, 10);
        ^~~~~~~~~~~~~~~~
./tests/fail_arguments.spl:16:15: error: Cannot unify types Int and Bool
        function(10, 10);
                     ^~
./tests/fail_arguments.spl:16:15: note: Type Int inferred here:
        function(10, 10);
                     ^~
./tests/fail_arguments.spl:4:24: note: Type Bool inferred here:
[Int] function (Int n, Bool b, [Int] l)
                       ^~~~
./tests/fail_arguments.spl:17:2: error: Too many arguments given, 4 given, but 3 expected.
        function(10, [], [], []);
        ^~~~~~~~~~~~~~~~~~~~~~~~
./tests/fail_arguments.spl:17:15: error: Cannot unify types Bool and [a33]
        function(10, [], [], []);
                     ^~
./tests/fail_arguments.spl:4:24: note: Type Bool inferred here:
[Int] function (Int n, Bool b, [Int] l)
                       ^~~~
./tests/fail_arguments.spl:17:15: note: Type [a33] inferred here:
        function(10, [], [], []);
                     ^~
*** Typing failed (but we try to continue!)
// function :: [Int](Int, Bool, [Int])
[Int] function/*6*/(Int n/*8*/, Bool b/*9*/, [Int] l/*10*/){
        print/*0*/(n/*8*/);
        print/*0*/(b/*9*/);
        print/*0*/(l/*10*/);
        return l/*10*/;
```

```
}

// main :: Void()
Void main/*7*/(){
        // list :: [Int]
        [Int] list/*11*/ = 1 : (2 : (3 : []));
        function/*6*/(10, 10);
        function/*6*/(10, [], [], []);
}
*** Done


*** Compiling ./tests/fail_empty_list.spl ***
// Typing error, t == Int == Bool is not possible

[t] x = [];
[t] y = []; // Same type as x

Void main(){
        print(1:x);    // => x is of type [Int]
        print(True:y); // => y is of type [Bool] => error
}
*** Done reading file
*** Lexing is done
*** Parsing succeeded
*** Scoping succeeded
./tests/fail_empty_list.spl:8:13: error: Cannot unify types Bool and Int
        print(True:y); // => y is of type [Bool] => error
                   ^
./tests/fail_empty_list.spl:8:8: note: Type Bool inferred here:
        print(True:y); // => y is of type [Bool] => error
              ^~~~
./tests/fail_empty_list.spl:7:8: note: Type Int inferred here:
        print(1:x);    // => x is of type [Int]
              ^
./tests/fail_empty_list.spl:3:2: error: Type mismatch. Expected type a0 declared here:
[t] x = [];
 ^
./tests/fail_empty_list.spl:7:8: note: Actual type Int inferred here:
        print(1:x);    // => x is of type [Int]
              ^
./tests/fail_empty_list.spl:4:2: error: Type mismatch. Expected type a0 declared here:
[t] y = []; // Same type as x
 ^
./tests/fail_empty_list.spl:7:8: note: Actual type Int inferred here:
        print(1:x);    // => x is of type [Int]
              ^
*** Typing failed (but we try to continue!)
// x :: [a0]
[t] x/*6*/ = [];

// y :: [a0]
[t] y/*7*/ = [];

// main :: Void()
Void main/*8*/(){
        print/*0*/(1 : x/*6*/);
        print/*0*/(True : y/*7*/);
}
*** Done


*** Compiling ./tests/fail_identifier_errors.spl ***
// multiple errors: Undeclared identifiers + redaclaration

Int bar = 5;
Int foo = 6;

// Undefined identifiers
Int blaat = goo + baz * blaat(fool);

// Redeclaration
Int blaat(){
        empty([]);
        return bat + foo + blaat;
}

*** Done reading file
*** Lexing is done
*** Parsing succeeded
./tests/fail_identifier_errors.spl:10:5: error: Redeclaration of identifier "blaat"
Int blaat(){
    ^~~~~
./tests/fail_identifier_errors.spl:7:5: note: Previous declaration here:
Int blaat = goo + baz * blaat(fool);
    ^~~~~
./tests/fail_identifier_errors.spl:7:13: error: Undeclared identifier "goo"
Int blaat = goo + baz * blaat(fool);
            ^~~
./tests/fail_identifier_errors.spl:4:5: note: Did you mean "foo"?
Int foo = 6;
    ^~~
./tests/fail_identifier_errors.spl:7:19: error: Undeclared identifier "baz"
Int blaat = goo + baz * blaat(fool);
                  ^~~
./tests/fail_identifier_errors.spl:3:5: note: Did you mean "bar"?
Int bar = 5;
    ^~~
./tests/fail_identifier_errors.spl:7:31: error: Undeclared identifier "fool"
Int blaat = goo + baz * blaat(fool);
                              ^~~~
./tests/fail_identifier_errors.spl:4:5: note: Did you mean "foo"?
Int foo = 6;
    ^~~
./tests/fail_identifier_errors.spl:11:2: error: Undeclared identifier "empty"
        empty([]);
        ^~~~~
./tests/fail_identifier_errors.spl:11:2: note: Did you mean "isEmpty"?
```

```
./tests/fail_identifier_errors.spl:12:9: error: Undeclared identifier "bat"
        return bat + foo + blaat;
               ^~~
./tests/fail_identifier_errors.spl:3:5: note: Did you mean "bar"?
Int bar = 5;
    ^~~
*** Scoping failed (but we try to continue!)
./tests/fail_identifier_errors.spl:10:5: error: Identifier is unknown: blaat
Int blaat(){
    ^~~~~
./tests/fail_identifier_errors.spl:10:5: note: This is probably caused by earlier errors
*** Stopping due to fatal error


*** Compiling ./tests/fail_void_no_return.spl ***
// multiple errors: Using Void and non-returning function foo.

Void baz(Int n){
        {}
}

Int foo(){
        if(True){}
}

Void main(){
        print(baz(5));
}
*** Done reading file
*** Lexing is done
*** Parsing succeeded
*** Scoping succeeded
./tests/fail_void_no_return.spl:7:1: error: Cannot unify types Int and Void
Int foo(){
^
./tests/fail_void_no_return.spl:7:1: note: Type Int inferred here:
Int foo(){
^~~
./tests/fail_void_no_return.spl:7:1: note: Type Void inferred here:
Int foo(){
^
./tests/fail_void_no_return.spl:7:1: error: Type mismatch. Expected type Int declared here:
Int foo(){
^~~
./tests/fail_void_no_return.spl:7:1: note: Actual type Void inferred here:
Int foo(){
^
./tests/fail_void_no_return.spl:12:2: error: Using void
        print(baz(5));
        ^~~~~~~~~~~~
./tests/fail_void_no_return.spl:3:1: note: Type Void inferred here:
Void baz(Int n){
^~~~
*** Typing failed (but we try to continue!)
// baz :: Void(Int)
Void baz/*6*/(Int n/*9*/){
        {}
}

// foo :: Int()
Int foo/*7*/(){
        if(True){
        }
}

// main :: Void()
Void main/*8*/(){
        print/*0*/(baz/*6*/(5));
}
*** Done


*** Compiling ./tests/pass_merge_sort.spl ***
// Jascha  Neutelings (s0610054)
// Passing example

Int length([t] list)
{
        if (isEmpty(list))
                return 0;
        else
                return 1 + length(tail(list));
}

[t] append([t] list, t value)
{
        if (isEmpty(list))
                return value : [];
        else
                return head(list) : append(tail(list), value);
}

([t],[t]) split_aux([t] list, Int n)
{
        ([t],[t]) tl = split(tail(list), n - 1);
        return (head(list) : fst(tl), snd(tl));
}

([t],[t]) split([t] list, Int n)
{
        if (isEmpty(list) || n == 0)
                return ([], list);
        else
                return split_aux(list, n);
}

[Int] merge_sort_aux([Int] m)
{
```

```
        Int middle = length(m) / 2;
        ([Int],[Int]) parts = split(m, middle);
        [Int] left = fst(parts);
        [Int] right = snd(parts);

        // recursively call merge_sort() to further split each sublist
        // until sublist size is 1
        left = merge_sort(left);
        right = merge_sort(right);
        // merge the sublists returned from prior calls to merge_sort()
        // and return the resulting merged sublist
        return merge(left, right);
}

[Int] merge_sort([Int] m)
{
        // if list size is 1, consider it sorted and return it
        if (isEmpty(m) || isEmpty(tail(m)))
                return m;
        // else list size is > 1, so split the list into two sublists
        return merge_sort_aux(m);
}

[Int] merge([Int] left, [Int] right)
{
        [Int] result = [];
        while (!isEmpty(left) || !isEmpty(right))
        {
                if (!isEmpty(left) && !isEmpty(right))
                {
                        if (head(left) <= head(right))
                        {
                                result = append(result, head(left));
                                left = tail(left);
                        }
                        else
                        {
                                result = append(result, head(right));
                                right = tail(right);
                        }
                }
                else if (!isEmpty(left))
                {
                        result = append(result, head(left));
                        left = tail(left);
                }
                else if (!isEmpty(right))
                {
                        result = append(result, head(right));
                        right = tail(right);
                }
        }
        return result;
}

Void main()
{
        [Int] list = 8 : 10 : 3 : 7 : 9 : 6 : 4 : 1 : 2 : 5 : [];
        print(merge_sort(list));
}
*** Done reading file
*** Lexing is done
*** Parsing succeeded
*** Scoping succeeded
*** Typing succeeded
// length :: forall a0 . Int([a0])
Int length/*6*/([t] list/*14*/){
        if(isEmpty/*1*/(list/*14*/))
                return 0;
        else
                return 1 + length/*6*/(tail/*3*/(list/*14*/));
}

// append :: forall a1 . [a1]([a1], a1)
[t] append/*7*/([t] list/*15*/, t value/*16*/){
        if(isEmpty/*1*/(list/*15*/))
                return value/*16*/ : [];
        else
                return head/*2*/(list/*15*/) : append/*7*/(tail/*3*/(list/*15*/), value/*16*/);
}

// split_aux :: forall a2 . ([a2], [a2])([a2], Int)
([t], [t]) split_aux/*8*/([t] list/*17*/, Int n/*18*/){
        // tl :: ([a2], [a2])
        ([t], [t]) tl/*19*/ = split/*9*/(tail/*3*/(list/*17*/), n/*18*/ - 1);
        return (head/*2*/(list/*17*/) : fst/*4*/(tl/*19*/), snd/*5*/(tl/*19*/));
}

// split :: forall a3 . ([a3], [a3])([a3], Int)
([t], [t]) split/*9*/([t] list/*20*/, Int n/*21*/){
        if(isEmpty/*1*/(list/*20*/) || (n/*21*/ == 0))
                return ([], list/*20*/);
        else
                return split_aux/*8*/(list/*20*/, n/*21*/);
}

// merge_sort_aux :: [Int]([Int])
[Int] merge_sort_aux/*10*/([Int] m/*22*/){
        // middle :: Int
        Int middle/*23*/ = length/*6*/(m/*22*/) / 2;
        // parts :: ([Int], [Int])
        ([Int], [Int]) parts/*24*/ = split/*9*/(m/*22*/, middle/*23*/);
        // left :: [Int]
        [Int] left/*25*/ = fst/*4*/(parts/*24*/);
        // right :: [Int]
        [Int] right/*26*/ = snd/*5*/(parts/*24*/);
        left/*25*/ = merge_sort/*11*/(left/*25*/);
        right/*26*/ = merge_sort/*11*/(right/*26*/);
        return merge/*12*/(left/*25*/, right/*26*/);
}

// merge_sort :: [Int]([Int])
[Int] merge_sort/*11*/([Int] m/*27*/){
        if(isEmpty/*1*/(m/*27*/) || isEmpty/*1*/(tail/*3*/(m/*27*/)))
                return m/*27*/;
        return merge_sort_aux/*10*/(m/*27*/);
}

// merge :: [Int]([Int], [Int])
[Int] merge/*12*/([Int] left/*28*/, [Int] right/*29*/){
        // result :: [Int]
        [Int] result/*30*/ = [];
        while((!isEmpty/*1*/(left/*28*/)) || (!isEmpty/*1*/(right/*29*/))){
                if((!isEmpty/*1*/(left/*28*/)) && (!isEmpty/*1*/(right/*29*/))){
                        if(head/*2*/(left/*28*/) <= head/*2*/(right/*29*/)){
                                result/*30*/ = append/*7*/(result/*30*/, head/*2*/(left/*28*/));
                                left/*28*/ = tail/*3*/(left/*28*/);
                        } else {
                                result/*30*/ = append/*7*/(result/*30*/, head/*2*/(right/*29*/));
                                right/*29*/ = tail/*3*/(right/*29*/);
                        }
                } else
                        if(!isEmpty/*1*/(left/*28*/)){
                                result/*30*/ = append/*7*/(result/*30*/, head/*2*/(left/*28*/));
                                left/*28*/ = tail/*3*/(left/*28*/);
                        } else
                                if(!isEmpty/*1*/(right/*29*/)){
                                        result/*30*/ = append/*7*/(result/*30*/, head/*2*/(right/*29*/));
                                        right/*29*/ = tail/*3*/(right/*29*/);
                                }
        }
        return result/*30*/;
}

// main :: Void()
Void main/*13*/(){
        // list :: [Int]
        [Int] list/*31*/ = 8 : (10 : (3 : (7 : (9 : (6 : (4 : (1 : (2 : (5 : []))))))))));
        print/*0*/(merge_sort/*11*/(list/*31*/));
}
*** Done


*** Compiling ./tests/pass_parser.spl ***
// Basic tests for associativity and prioities of operators

Int x = 5-4-3*9/4/3+2;          // - is left assoc
[Int] list = 1:2:3:4:[];        // : is right assoc
[Int] list2 = 1+2*7 : list;

Bool y = x == 4 || x > 5 || x != 0;
[Bool] list3 = (x == 6) : True : y : [];

*** Done reading file
*** Lexing is done
*** Parsing succeeded
*** Scoping succeeded
*** Typing succeeded
// x :: Int
Int x/*6*/ = ((5 - 4) - (((3 * 9) / 4) / 3)) + 2;

// list :: [Int]
[Int] list/*7*/ = 1 : (2 : (3 : (4 : [])));

// list2 :: [Int]
[Int] list2/*8*/ = (1 + (2 * 7)) : list/*7*/;

// y :: Bool
Bool y/*9*/ = (x/*6*/ == 4) || ((x/*6*/ > 5) || (x/*6*/ != 0));

// list3 :: [Bool]
[Bool] list3/*10*/ = (x/*6*/ == 6) : (True : (y/*9*/ : []));
*** Done


*** Compiling ./tests/pass_polymorphism.spl ***
// Passes, note that those t's are different

[t] x = [];

[t] list_id([t] list){
        return list;
}

t id(t x){
        return x;
}

Void main(){
        print(id(list_id(x)));
}
*** Done reading file
*** Lexing is done
*** Parsing succeeded
./tests/pass_polymorphism.spl:9:8: warning: "x" shadows previous declaration.
t id(t x){
       ^
./tests/pass_polymorphism.spl:3:5: note: Previous declaration was a global:
[t] x = [];
    ^
*** Scoping succeeded
*** Typing succeeded
// x :: [a0]
[t] x/*6*/ = [];

// list_id :: forall a1 . [a1]([a1])
```

```
[t] list_id/*7*/([t] list/*10*/){
        return list/*10*/;
}

// id :: forall a2 . a2(a2)
t id/*8*/(t x/*11*/){
        return x/*11*/;
}

// main :: Void()
Void main/*9*/(){
        print/*0*/(id/*8*/(list_id/*7*/(x/*6*/)));
}
*** Done


*** Compiling ./tests/pass_reverse.spl ***
// Simple example, also using lists of lists

[t] reverse([t] list)
{
        [t] accu = [];
        while(!isEmpty(list))
        {
                accu = head(list) : accu;
                list = tail(list);
        }
        return accu;
}

Void main(){
        print(reverse(1:2:3:[]));
        print(reverse((1:2:3:[]):[]));
}
*** Done reading file
*** Lexing is done
*** Parsing succeeded
*** Scoping succeeded
*** Typing succeeded
// reverse :: forall a0 . [a0]([a0])
[t] reverse/*6*/([t] list/*8*/){
        // accu :: [a0]
        [t] accu/*9*/ = [];
        while(!isEmpty/*1*/(list/*8*/)){
                accu/*9*/ = head/*2*/(list/*8*/) : accu/*9*/;
                list/*8*/ = tail/*3*/(list/*8*/);
        }
        return accu/*9*/;
}

// main :: Void()
Void main/*7*/(){
        print/*0*/(reverse/*6*/(1 : (2 : (3 : []))));
        print/*0*/(reverse/*6*/((1 : (2 : (3 : []))) : []));
}
*** Done


*** Compiling ./tests/warn_shadowing.spl ***
// Correct code, but shadows identifiers (=> warnings)

Void main(){
        print(y(x(y(5))));
}

x x(x x){
        x y = x;
        return x;
}

y y(y y){
        y y = x(y);
        return y;
}
*** Done reading file
*** Lexing is done
*** Parsing succeeded
./tests/warn_shadowing.spl:7:7: warning: "x" shadows previous declaration.
x x(x x){
      ^
./tests/warn_shadowing.spl:7:3: note: Previous declaration was a global:
x x(x x){
  ^
./tests/warn_shadowing.spl:8:4: warning: "y" shadows previous declaration.
        x y = x;
          ^
./tests/warn_shadowing.spl:12:3: note: Previous declaration was a global:
y y(y y){
  ^
./tests/warn_shadowing.spl:12:7: warning: "y" shadows previous declaration.
y y(y y){
      ^
./tests/warn_shadowing.spl:12:3: note: Previous declaration was a global:
y y(y y){
  ^
./tests/warn_shadowing.spl:13:4: warning: "y" shadows previous declaration.
        y y = x(y);
          ^
./tests/warn_shadowing.spl:12:7: note: Previous declaration was used as argument to function:
y y(y y){
      ^
*** Scoping succeeded
*** Typing succeeded
// main :: Void()
Void main/*6*/(){
        print/*0*/(y/*8*/(x/*7*/(y/*8*/(5))));
}
```

```
// x :: forall a0 . a0(a0)
x x/*7*/(x x/*9*/){
        // y :: a0
        x y/*10*/ = x/*9*/;
        return x/*9*/;
}

// y :: forall a1 . a1(a1)
y y/*8*/(y y/*11*/){
        // y :: a1
        y y/*12*/ = x/*7*/(y/*11*/);
        return y/*12*/;
}
*** Done

joshua@192:~/Documents/Code/splang/midreport/tests (master)$
```