Christopher Pac
N-13242624
cpp270@nyu.edu

Compiler Construction
Spring 2015
Project Milestone 2

TABLE OF CONTENTS

1 Introduction

This document will cover the documentation requirement part specified in the Project Milestone 2. It will explain disasters that were encountered during the SDD implementation and the solutions that were tried but only succeeded half the time. The document will first cover the general differences between the provided SDD definitions and what has been implemented. These differences are minor and the code looks very close to what has been provided.

This document is divided into sections that naturally follow the project. The introduction section provides information about the general approach that was taken while working on the SDD, the computing environment that was used, and outside sources that were used while working on this project.

The rest of this document covers the implementation and issues, some general problems that were encountered with HACS, lessons learned, and all the tests that were done on the SDD. The test section will also described all the extra attached files and how they were used.

1.1 General Approach to the Project

I have followed the provided SDD solution exclusively and only used my SDD solution for learning purposes. In my project solution I have also used the provided base file as a starting point and not my original parser implementation. I found the parser provided to be a little cleaner and in general I have more confidence in it.

I have encountered numerous problems that have taken hours to figure out and were relatively simple once discovered and I have also come upon problems that seem to defy logic.

1.2 Environment

- Computing environment
  - Linux energon1.cims.nyu.edu
- HACS version
  - HACS 1.0.12

1.4 Credits

I did minimal work with Martha Poole. We shared information about bugs that were encountered and she also shared only the Sse scheme definition with me. It turned out that her logic was identical to mine and the problem I was having was in a different place. I have also received help from Jose that literally unable me to have a working project. Thanks Jose.

2 Implementation And Issues

This section will cover all the major differences between my implementation and the reference SDDs. I will then briefly describe the layout of the source code and finish by analyzing the issues that I have encountered.

I will not provide detail information about the SDDs as they are cover thoroughly in the reference material. Almost all of the logic in the source code is reflected in the reference and I will clarify the few complicated lines of code in the source code itself. I would like to apologize ahead of time for not wrapping long lines of code but I found that style to be more easily readable.

2.1 Major Reference SDD Implementation Differences

As I have mentioned before, my implementation follows very closely to the provided SDDs. The two major differences are that my type synthesized attribute *t* is not a vector but a single element and I have not used the *ns* attribute and instead synthesized both the map and a list of type descriptors. These differences did not make much of a difference in the actual flow and logic.

The reason for using a vector for the type attribute is only to make it convenient to synthesizing a list of function/method type parameters. In all other places it is sufficient to have a single value for the type attribute. I have used a separate synthesized list on the expression productions to create the vector of types and therefore my expression production has three attributes: *t, ps, e*.

The reference solution uses *ps* type vector and a *ns* name vector to create the name to type map. I have replaced the *ns* with a map thus in all the places where *ns* is used I just synthesize a map represented by the *mm* attribute. So, *mm* replaces *ns*.

The function names are the same as in the reference except that the extend function is called TDMapAppend.

2.2 Code Layout

The first part of the code has the provided parser implementation, this is followed by all the semantic structures and functions that are used This is followed by the wire up code, that does the check, and finally the main implementation. The main implementation is split between the synthesized attribute section and the inherited section.

2.3 Encountered Issues

2.3.1

The first issues stemmed from misunderstanding how synthesized attributes are handled when an inherited attribute is passed down. For some reason I thought that HACS will somehow automatically synthesize the attribute when the syntactic definition is provided. For example having provided the following:

```
sort Statement | ↑ok ;
⟦ ; ⟧ ↑ok(True);
```

I expected HACS to get it without providing the corresponding Se defined scheme for that production. I thought that the only reason to write the scheme rule is when one wants to override the attribute. I missed the full reason for having ↑#syn which captures all the attributes and more importantly when used on the left side it causes the attributes to be synthesized. Thus the solution to the problem that plagued me for a whole day was to simply have `Se( ⟦ ; ⟧ ↑#syn) → ⟦ ; ⟧ ↑#syn;` as the corresponding rule. I figured out pretty early that I needed the corresponding rule it is just that I thought I would have to set the attribute for ever rule instead of having it automatically synthesized based on the syntactic definitions. Thanks Jose for the help! Once I realized that all I needed was these simple functions most of my implementation started to work.

2.3.2

I've encountered problems with searching for keys in a map where the keys were set in a different scope of the AST. The key is a special HACS symbol and it seems to append a numeric value to the key. Thus when I set RETURN and THIS in one scope I was not able to find it in another. I believe my logic when setting those values is correct and because other variable definitions work means that my underlying structures and search functions work.

I think that my the *nds* attribute is correctly synthesized and passed to the inherited environment. This is due to the fact that I am able to call functions from another function. This function calling would not be possible if the *nds* attribute was not correctly done.

Furthermore, this same bug manifests itself for methods and class attributes since they use ID and not Name. This causes the ID not to be found later in a lower scope.

The solution to the problem could be hacs string to symbol conversion which would enable setting arbitrary string values. A solution suggested by Jose is to have the **return** and **this** passed down through custom inherited attribute. I have implemented the inherited attribute workaround for to the **return** problem. The solution for **this** would be the same.

Update

I have been able to resolve both problems by creating a MapKey sort and defining *Return* and *This* constants.

2.3.3

My most recent problem makes me question my sanity. It is possible that it is related to the scoping problems however I find that unlikely. Due to this bug only functions with one or zero parameters can be called. However, functions with multiple parameters can still be declared and those parameters will be correctly checked inside the function. This code works correctly

```
function void foobar(int i, string s) { var string x; x = s + i; }
```

However, this function would fail if it was called form another function due to two parameters.

The problem seems to be caused by HACS's scheme not returning exactly whats being passed when requested.

The following code snippet produced correct results while testing:

```
↑t( DoCall(#t1 , TDListConst(TDType(⟦ int ⟧), TDListEmpty)) )
```

These following two code lines while logically equivalent failed to produce correct results:
The TDListJustRet function just returns what was passed in yet it fails.

```
↑t( DoCall(#t1 , TDListJustRet(TDListConst(TDType(⟦ int ⟧),
TDListEmpty))) )
```

The TDListInt scheme just returns the structure that worked above.

```
↑t( DoCall(#t1 , TDListInt() ) )
```

Where the definition of those two functions are:

```
sort TDList | TDListConst(TD, TDList) | TDListEmpty ;
| scheme TDListInt()
| scheme TDListJustRet(TDList)
;

TDListInt()  → TDListConst(TDType(⟦ int ⟧), TDListEmpty);
TDListJustRet(#)  → # ;
```

One function creates and returns exactly the same structure that worked successfully and the other just returns back what was passed in. Professor suggested that I should inspect my functions for the *default* keyword and not use it for computed values. I have done so and the problem still persists. I do not understand why the same value would worked when directly passed to the DoCall function but failed when just returned from another function. I do not believe its the scoping issues since a function with one parameter works and that type was synthesized on demand.

This problem deals with one of hacs pitfalls for the beginners. Functions can ignore comparisons if the values need to be computed. Therefore, one should only compare computed values and not values against nested functions.

Update

I have resolved this issue by synthesizing a type vector instead of calculating one.

2.3.4

I have encountered yet another frustrating issues where I'm not able to check if a class name is defined. This should be very similar to being able to check if called function was previously defined. My function checking and calling works properly but I am not able to lookup the class name. Both the class name and function name are in the same map where the function names are located. I have converted all my Names to IDs and the problem still persists.

I have been able to find that my TDMapDefined is one of the main culprits. This function is identical, except for return type, to my TDMapLookup function. TDMapLookup works the other does not.

Update

I have been able to resolve the issue of not being able to do the lookup. There were three problems that prevented my code from working properly. Two of them were very simple. The first problem was that I did not pass the environment to a recursive function explicitly. This is a potential hacs bug as it should not be necessary to pass these around if they are not used. My second problem was a simple initializing of TDType where I was passing another TDType inside its constructor.

The last problem had to do with resolving hacs stubbornness to pick the default rule instead of calculating a passed in function. This was fixed through fully expending the rules and recursion.

3 General Issues

There are two main problems when working with HACS that make it somewhat frustrating and tedious. The lack of functionality that would permit printing at any point causes long ad-hoc work arounds. I found myself using the *ok* attribute as a debugging tool during this project. I know that functionality to print the current state exists and is done when error is encountered. It would be nice to be able to trigger that printing through a keyword.

The second issue is the long compilation times and when this is coupled with the ad-hoc debugging it causes a lot of down time. I have a feeling that Java is to blame for the slowness.

4 Lessons Learned

In this section I will cover some general lessons that I have learned while working on this project.

4.1 SDD Dependencies

Establishing the proper dependencies between attributes is crucial when coding SDDs. If this first step is not done meticulously it will cause major issues later on. The most obvious issue will be that of circular dependency where a program can just loop or hang. Carefully planning the attributes also has the added benefit of keeping the number of attributes to minimum while computing what's necessary. In my SDD I had the *ok* attribute defined on almost every production. In my SDD I have also not implemented the top level scope definitions correctly and have not passed them down to the inherited environment. Now I understand how that two-pass of first synthesizing and then inheriting can be implemented.

4.2 Side effects

Coding SDDs is equivalent to coding in functional paradigm. I did not grasped that fully when I have initially designed my SDDs. The environment attribute, for example, is not modified but simply extend (i.e. new one is created) and passed down to appropriate production. My environment was a global stack that would was logically equivalent but its modifications caused side effects, that is the same stack was uses and modified from all places. During the implementation of this project a lot of functional programming concepts have started to come back to me. I have previously done some programming in ML and Lisp and the ideas are naturally similar here in HACS.

4.3 HACS

After this project I feel like I have a pretty good handle on working with hacs. I understand how it handles synthesized and inherited attributes and how the AST is used to connect it all together. There are still some hacs nuances that elude me; I have described one in 2.3.3.

The semantic structures are very tempting to use but there seems to be quite a few pitfalls that one needs to be aware. This especially applies to calculated values being use for comparisons. I believe that if I was redoing this project I would try to use the syntactic definitions for my structures.

5 Testing

All tests were done in the environment specified in section 1.2 Environment. I have tested my solution against custom test and the samples provided in project milestone 1. Below is just a sample of the tests that I have conducted. All together, excluding the sample and test files, I have 89 unit test that are included in the `Pr2UnitTests` file. The file has the string that is passed to the —term and the correct output. The `RunTests` script executes all the unit tests in the Pr2UnitTests file. The script takes -v (optional) and the hacs run file as input parameters. If the -v is not provided then only errors are printed. I have included both files in my project submission.

5.1 Command Line Testing

In this section I will list simple tests done on the command line that illustrate a correctly working type checking. I have done here checks that correctly pass the test and correctly fail the test. The “Error” tests are important to illustrate that there is actual checking being done. Thus here I tried to provide a pass test and a mirror fail test.

5.1.1
Undefined variable check.

```
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main() { var int
n1 ; { var int n2; {n1 = n1 + n2; } }}'
OK
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main() { var int
n1 ; { var int n2; {n1 = n1 + nXXXX; } }}'
ERROR
```

5.1.2
Undefined type check.

```
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main(int i, string
s) { }'
OK
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main(int i, string
s, foobar x) { }'
ERROR
```

5.1.3
Check for adding integer and string and subtracting them. Subtraction should error.

```
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main(int i, string
s) { var string x; x = s + i; }'
OK
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main(int i, string
s) { var string x; x = s - i; }'
ERROR
```

5.1.4
Tests for assignment ‘+=‘ and simple literals.

```
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main(int i, string
s) { var string x; x = s + i; }'
OK
// cant subtract here
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main(int i, string
s) { var string x; x = s - i; }'
ERROR
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main(int i, string
s) { var string x; x += i; }'
OK
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main(int i, string
s) { var string x; x += 20; }'
OK
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main(int i, string
s) { var int Y; Y += 20; }'
OK
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main(int i, string
s) { var int Y; Y += i; }'
OK
// += int and string and assign to int
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main(int i, string
s) { var int Y; Y += s; }'
ERROR
```

5.1.5
Simple boolean checks. Second statement fails cause int cant be assigned to bool.

```
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main(boolean b) {var
int x; var int y; b = x > y; }'
OK
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main(boolean b) {var
int x; var int y; b = x; }'
ERROR
```

```
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main(boolean b) {var
int x; var int y; b = "hello" == "world"; }'
```
OK

5.1.6
Checking function parameter passing and function return type.

```
$ ./Pr2ChrisPac.run --scheme=Check --term='function int foobar(string s) { }
function void main(int b) { b = foobar("Hello"); }'
OK
```
// Errors cause foobar takes a string.
```
$ ./Pr2ChrisPac.run --scheme=Check --term='function int foobar(string s) { }
function void main(int b) { b = foobar( 500 ); }'
ERROR
```
// Errors cause foobar returns an int not string
```
$ ./Pr2ChrisPac.run --scheme=Check --term='function int foobar(string s) { }
function void main(string s) { s = foobar( s ); }'
ERROR
$ ./Pr2ChrisPac.run --scheme=Check --term='function int foobar(string s) { }
function void main(string s) { var int i; {i = foobar( s );} }'
OK
```
// errors cause foobar returns int not string (Done in nested scope)
```
$ ./Pr2ChrisPac.run --scheme=Check --term='function int foobar(string s) { }
function void main(string s) { var int i; {s = foobar( s );} }'
ERROR
```

5.1.7
Tests for the if statement and nested scopes.

```
$ ./Pr2ChrisPac.run --scheme=Check --term='function int foobar(string s) { }
function void main(string s) { var boolean b; var int i; { if (b) {i =
foobar( s );}} }'
OK
```

// If errors cause b was changed from boolean to string
```
$ ./Pr2ChrisPac.run --scheme=Check --term='function int foobar(string s) { }
function void main(string s) { var string b; var int i; { if (b) {i = foobar(
s );}} }'
ERROR
$ ./Pr2ChrisPac.run --scheme=Check --term='function int foobar(string s) { }
function void main(string s) { var boolean b; var int i; { if (5>6) {i =
foobar( s );}} }'
OK
```

5.1.8
Simple while test.

```
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main(int i ) { while
(5>6) { i += 1; }}'
OK
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main(int i ) { while
(5>6) { i += "hello"; }}'
ERROR
```

5.1.9

While test with more complicated expression.

```
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main(boolean b )
{var int i; while ( i > 6 || b ) { i += 1; }}'
OK
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main(boolean b )
{var int i; while ( i > "hello" || b ) { i += 1; }}'
ERROR
```

5.1.10

More while tests.

```
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main(boolean b, int
i, int j ) {var int i; while ( i > j || b ) { i += 1; }}'
OK
```

// Error because of boolean > int

```
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main(boolean b, int
i, int j ) {var int i; while ( b > j || b ) { i += 1; }}'
ERROR
```

5.1.11

Variable shadowing.

```
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main( ) { var string
x; { var int x; x = 6 ;}}'
OK
$ ./Pr2ChrisPac.run --scheme=Check --term='function void main( ) { var string
x; { var int x; x = "hello" ;}}'
ERROR
```

5.1.12

Functions with multiple parameters.

```
cpp270@energon1[Tests]$ ./Pr2ChrisPac.run --scheme=Check --term='function int
foobar (string h, int s, string s2) { } function int main(string s ) { var
int z; foobar("Hello", z, s); }'
OK
cpp270@energon1[Tests]$ ./Pr2ChrisPac.run --scheme=Check --term='function int
foobar (string h, int s, string s2) { } function int main(string s ) { var
int z; foobar("Hello", "Wrong Arg", s); }'
ERROR
```

5.1.13

Passing functions to functions.

```
cpp270@energon1[Tests]$ ./Pr2ChrisPac.run --scheme=Check --term='function int
foobar (string h, int i, string s2) { } function int main(string s ) { var
int z; foobar("Hello", main("k"), s); }'
OK
cpp270@energon1[Tests]$ ./Pr2ChrisPac.run --scheme=Check --term='function int
foobar (string h, int i, string s2) { } function string main(string s ) { var
int z; foobar("Hello", main("main returns string"), s); }'
ERROR
```

5.1.14

Using the return keyword in tested statements.

```
cpp270@energon1[Tests]$ ./Pr2ChrisPac.run --scheme=Check --term='function int
main( int x ) { var int i; if (x>7) {  { return x; }} }'
OK
cpp270@energon1[Tests]$ ./Pr2ChrisPac.run --scheme=Check --term='function int
main( int x ) { var int i; if (x>7) {  { return "Wrong"; }} }'
ERROR

// Shadowing and tested return.
cpp270@energon1[Tests]$ ./Pr2ChrisPac.run --scheme=Check --term='function int
main( int x ) { var string v; if (x>7) { var int v; v = 7; { return v; }} }'
OK

// Simple void return.
cpp270@energon1[Tests]$ ./Pr2ChrisPac.run --scheme=Check --term='function int
main( ) { return ; }'
ERROR
cpp270@energon1[Tests]$ ./Pr2ChrisPac.run --scheme=Check --term='function
void main( ) { return ; }'
OK
```

5.1.14

A comprehensive class, function, return, this, object literals test.

```
./Pr2ChrisPac.run --scheme=Check --term='class bar { int i1; string s2; int
f1(int v) { return i1;}} class foo { foo g;  int i; bar b; foo getC( string s
) { return this; } void putB(bar b) {}} function int compLen( string s)
{ return 5; } function void main() { var bar b; var foo f1; var foo f2;
f2=f1.getC("ok"); b={s2:"Hello"}; f1.putB(b); f1.i=compLen("ok");}'
OK
```

5.2 File Testing

For these test I used files that have more extended and slightly more complicated code. I will be attaching these files to my project.

5.2.1

This file test all the assignments and expression operations.

```
$ ./Pr2ChrisPac.run --scheme=Check samples/
OKSimpleTest_ExpOpsAndAssignment.jst
OK
```

5.2.2

This file test the same thing as 5.2.1 but at different scopes. Where some variables are declared and used in triple nested scope.

```
$ ./Pr2ChrisPac.run --scheme=Check samples/
OKAdvancedTest_ExpOpsAndAssignmentTripleNest.jst
OK
```

5.2.3

This file just test nested if and while statements.

```
$ ./Pr2ChrisPac.run --scheme=Check samples/OKAdvancedTest_ControlStats.jst
OK
```

5.2.4

This file test an undefined variable in a tested if statement.

```
$ ./Pr2ChrisPac.run --scheme=Check samples/FailSimpleTest_ControlStats.jst
ERROR
```

5.2.5

I have found a type error in the FastFib.jst script by running it through my type checker. I have corrected the error and included both FastFib.jst and CorrectedFastFib.jst. The error was on line 16 where the function was declared to return Pair class but instead was returning an integer.

```
./Pr2ChrisPac.run --scheme=Check samples/CorrectedFastFib.jst
OK
./Pr2ChrisPac.run --scheme=Check samples/FastFib.jst
ERROR
```

6 Outstanding Issues and Possible Solutions

~~6.1~~

Unable to search for **this** and **return** thus the test that use those keyword fail. I have previously discussed the underlying issue and possible solution in 2.3.2.

~~Partial Fix!~~

I have implemented a fix for so that **return** now works for functions. See test 5.1.14. I have used another inherited attribute *r* to pass down the type of the function and use that attribute later to check. This exact fix would also work for **this** keyword, however, classes are not properly working.

Fixed!

Both the **return** and **this** work correctly now. The **return** continues to use the inherited attribute. The **this** keyword is implemented as described in the SDD solution by extending the environment map with **this** key. I have tried the same solution that I used in fixing **this** for the **return** and it works. However, because I have already tested **return**, I decided to keep using inherited attribute.

~~6.2~~

Class method do not properly work. A simple method declaration returns an error. This could be related to my main issue of not being able to use classes as types. Furthermore, I did not have enough time to debug the class functionality. It successfully runs with just the class/variable/attribute definitions but fails when those are used.

Fixed!

I synthesized a map of member definitions and extended the environment on the way down to Members. This is similar to first synthesizing the *nds* and passing it into the inherited attribute of a Program.

~~6.3~~

Due to classes not working as expected I was not able to test the object literals. I believe the logic is correctly implemented barring some usual minor issues.

Fixed!

Object literals can now be assigned to a class variable. The object literal needs to have the same names and types as the class. However, it is not required to have all the members of a class in the object literal only a subset of them. This was a complicated issue to fix due to HACS not evaluating inner functions when it has an alternative choice. I have fixed it by fully expending the definitions and doing a recursive call on the *default* choice. This forced HACS to have to calculate the inner function.

~~6.4~~

Unable to use a defined class as a type. I have narrowed down the problem to my TDMapDefined function. It is a simple function that takes no calculated values and works almost exactly like TDMapLookup. So, the functionality is very similar to looking up function names in nested statements. However, TDMapDefined cause a crash if a class name/id is passed in. It is possible that there is an issues with the actual map.

Fixed!

The solution to the crash was very simple. It required that the environment was explicitly passed to the recursive helper function.

~~6.5~~

Unable to call functions with more than one parameter. I believe my underling logic for making this work is correct. I synthesize a type vector on Exp , Exp production and an empty vector otherwise. When this vector is empty I use a single type attribute to create the vector otherwise I use the vector list itself. However, for some reason a simple scheme return does not seem to work. I have described this very frustrating and confounding problem in 2.3.3.

The program does, however, check if multiple (more than one) formal parameters are correctly used in the function scope itself.

Fixed!

I recalculate a new vector of types just for this. It is not an elegant solution and if I had time I would rewrite how my synthesized *type* attribute works. There could also be a fix for my initial implementation but after many attempts I have not been able to make the old way work…

6.6

User tokens to fixed tokens comparisons are not working in hacs. I have followed Prof. Rose's instructions and he has also helped me directly write part of the DoMember function where the comparison is being done. It does't work. For this reason the functions *length/charCodeAt/substr* do not properly resolve to a type. I have setup a simple test where I created a scheme that takes an expression and checks if the ID is the same as “length”.

```
TestIDCompare(⟦ ⟨Expression#1⟩ . ⟨ID#2⟩ ⟧) → IsSameID(#2, ⟦ length ⟧);

IsSameID(#, #) → True ;
default IsSameID(#1, #2) → False ;
```

When I tested this by passing in *“ok”.length* it returned false.

```
./Pr2ChrisPac.run  --scheme=TestIDCompare --sort=Expression --term='"ok".length'
'$Print2-Pr2ChrisPac$Boolean'[Pr2ChrisPac$Boolean_False, 0]
```

The code in DoMember function properly checks for *length/charCodeAt/substr* and returns the correct type. If the user tokens to fixed tokens comparison works then I think my code will produce correct results.

7 HACS Bugs

7.1 `error` keyword

The `error` keyword does not behave as expected.

```
sort TDMap | TDMapConst(Name, TD, TDMap) | TDMapEmpty

| scheme CheckIfEmpty(TDMap)
;

CheckIfEmpty(TDMapEmpty) → error⟦empty⟧ ;
default CheckIfEmpty(#1) → #1;
```

In the example scheme provided above when an empty map is passed in, while using the scheme in other nested schemes, HACS does not abort with the error message. Instead HACS ‘fixes’ the problem by defining the error⟦empty⟧ as being part of the sort. Here’s an output from a command line run:

```
Warning in rule 'Pr2Base-Pr2Base$TDMap_CheckIfEmpty-1' (fixed): rule declares
fresh variables not used in contractum ([error_1])
OK
```

7.2 comment not ignored

The following comment was not ignored by HACS and caused it not to build the script.

```
/*
*** INPUT and OUTPUT to/from TestIDCompare -> IsSameID
*cpp270@energon1[Tests2]$ ./Pr2ChrisPac.run --scheme=TestIDCompare --sort=Expression --
term='"ok".length'
*« '$Print2-Pr2ChrisPac$Boolean'[Pr2ChrisPac$Boolean_False, 0] »
*/
```

I think » is the cause of the problem.