Ada Runtime Error Generator

Design Document

Date:18/12/2020

Student: Derry Brennan

Student number: C00231080

Supervisor: Chris Meudec

DECLARATION

Signature: Derry Brennan Date: 29/04/2021

Derry Brennan (Student)

C00231080 (Student Number)

The above declaration is confirmed by:

Signature: Chris Meudec Date: 29/04/2021

Dr. Christophe Meudec (Project Supervisor)

Table Of contents

Table of Figures

Abstract

“Ada is a state-of-the-art programming language that development teams worldwide are using for critical software: from microkernels and small-footprint, real-time embedded systems to large-scale enterprise applications, and everything in between.” [1] Used by the military, avionics and many other fields where safety is of critical importance.

With the reliance of safety in Ada it is pertinent to look into the ability to have run-time error free programs. An error that happens in the field could cause the loss of life or the destruction of property. My goal is to produce a prototype proof-of-concept tool which will be able to take Ada code and be able to tell the programmer if there is any possibility of runtime errors in their code and where. This document will be focused on my research in the different areas required for this project such as Ada itself, parsers and the different types of runtime errors.

1 Introduction

As technology expands throughout the world, taking control of complex tasks such as avionics and missile control, the need to make sure such software is free from as many errors as possible is of paramount importance. Programming languages have many forms of error checking in place already, with the integrated development environments having both semantic and syntactic errors being detected as the code is being written. But runtime errors are more difficult to find and not as much research and development has gone into the finding of such errors before.

Runtime errors such as division by zero, integer overflow and index out of bounds errors can cause a program to output unexpected results or to cease functioning entirely, neither of which is a good result, especially where lives are at stake.

Ada is a programming language that has its focus on safety and was seen as an ideal candidate to provide the testing ground for such a tool.

The Ada runtime error generator will be a tool that can be used by an Ada programmer who wishes to test their code for the presence of possible runtime errors.

This document will describe the process of the design of both the Ada runtime error generator and the Mika extension for Visual Studio Code.

2 Initial setup

2.1 Compilation

The first part of the process of understanding how to accomplish the task set out by the project was to compile the Ada parser used by the Mika tool.

There are two Flex files, ourxref.l and ada.l and two bison files ourxref.y and ada.y. The ourxref files need to be run through Flex and Bison respectively to produce output files used for compilation, these are lex.yy.c produced from ourxref.l and ourxref.tab.c and ourxref.tab.h produced from the ourxref.y file.

Once these output files have been produced they need to be added to a blank Visual Studio 2019 project. This is where some difficulties were encountered, see figure 15 below.

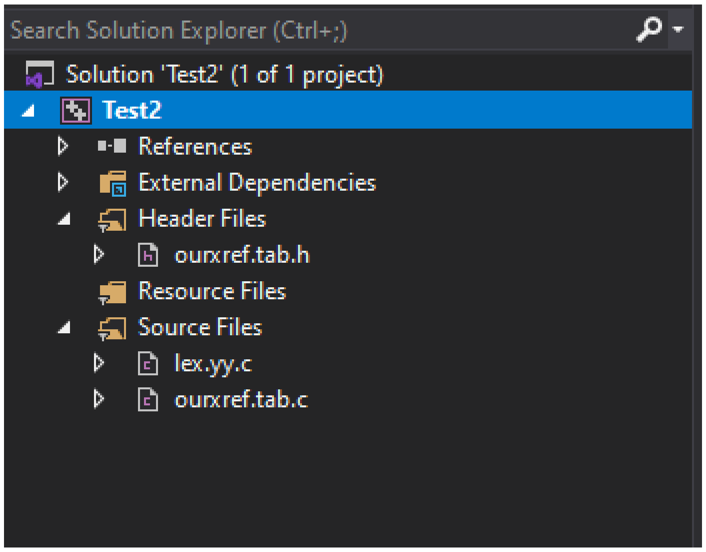
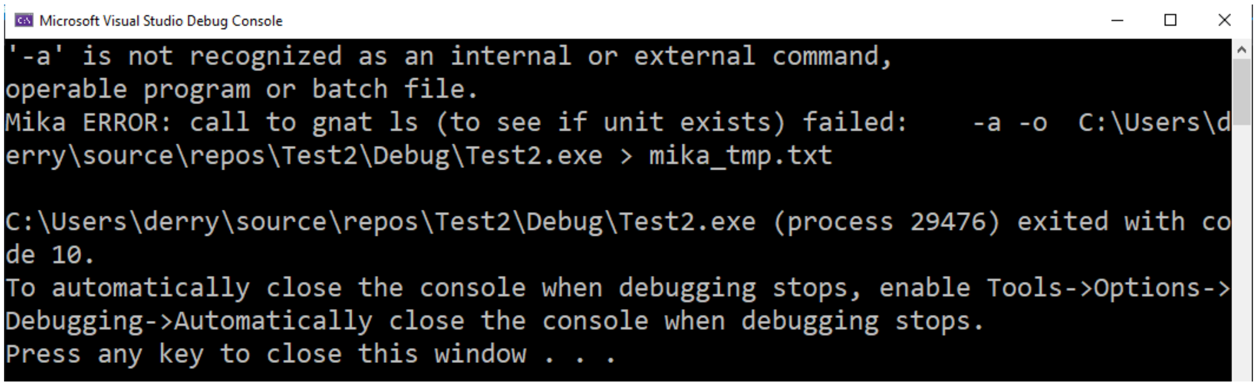


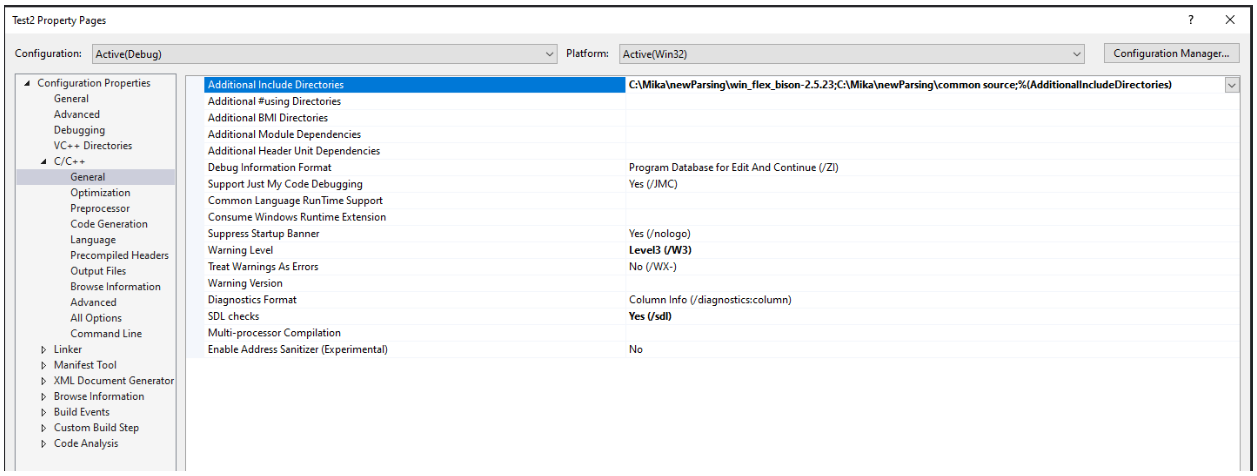
Figure 1 - Project with output files in VS2019

Upon executing this project the project was met with errors as shown in figure 16. A double check was done to make sure that the required folders were linked to the project's C/C++ include directories and the correct folders were indeed present as shown in figure 17.



```
'-a' is not recognized as an internal or external command,
operable program or batch file.
Mika ERROR: call to gnat ls (to see if unit exists) failed:    -a -o  C:\Users\d
erry\source\repos\Test2\Debug\Test2.exe > mika_tmp.txt

C:\Users\derry\source\repos\Test2\Debug\Test2.exe (process 29476) exited with co
de 10.
To automatically close the console when debugging stops, enable Tools->Options->
Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Figure 2 - Error received trying to execute the project



Configuration: Active(Debug) Platform: Active(Win32)

Additional Include Directories	C:\Mika\newParsing\win_flex_bison-2.5.23;C:\Mika\newParsing\common source;%(AdditionalIncludeDirectories)
Additional #using Directories	
Additional BMI Directories	
Additional Module Dependencies	
Additional Header Unit Dependencies	
Debug Information Format	Program Database for Edit And Continue (/ZI)
Support Just My Code Debugging	Yes (/JMC)
Common Language RunTime Support	
Consume Windows Runtime Extension	
Suppress Startup Banner	Yes (/nologo)
Warning Level	**Level3 (/W3)**
Treat Warnings As Errors	No (/WX-)
Warning Version	
Diagnostics Format	Column Info (/diagnostics:column)
SDL checks	**Yes (/sdl)**
Multi-processor Compilation	
Enable Address Sanitizer (Experimental)	No

Figure 3 - Linking the project to required files from mika

After a prolonged period of confusion the correct way of obtaining the needed executable files from the compilation of the project was determined. As is depicted in figure 18, right clicking on the solution and selecting build compiles the output files and generates the necessary files (figure 19). With this knowledge in hand the next step was to compile the ada.l and ada.y files in the same manner.

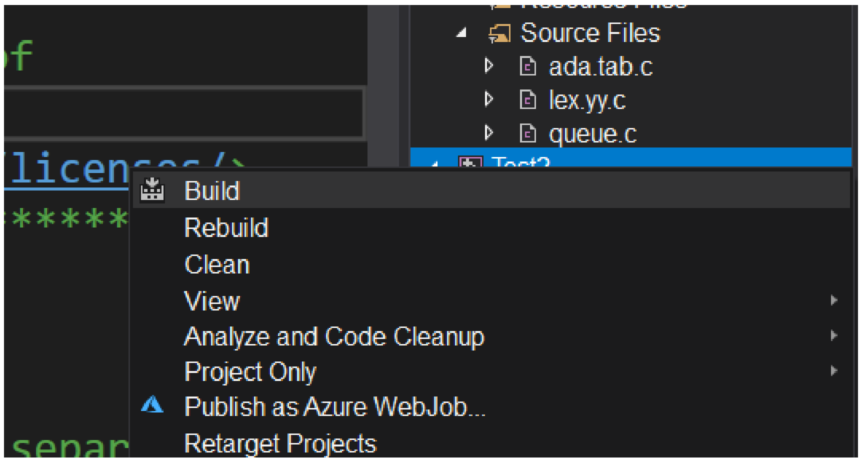
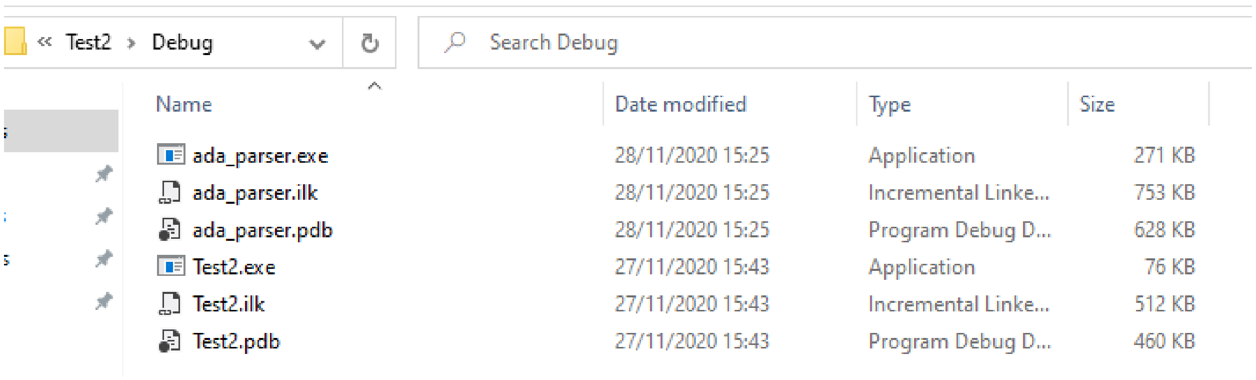


Figure 4 - the correct way to obtain executable file



Name	Date modified	Type	Size
ada_parser.exe	28/11/2020 15:25	Application	271 KB
ada_parser.ilk	28/11/2020 15:25	Incremental Linke...	753 KB
ada_parser.pdb	28/11/2020 15:25	Program Debug D...	628 KB
Test2.exe	27/11/2020 15:43	Application	76 KB
Test2.ilk	27/11/2020 15:43	Incremental Linke...	512 KB
Test2.pdb	27/11/2020 15:43	Program Debug D...	460 KB

Figure 5 - The output from compilation, with debug specified

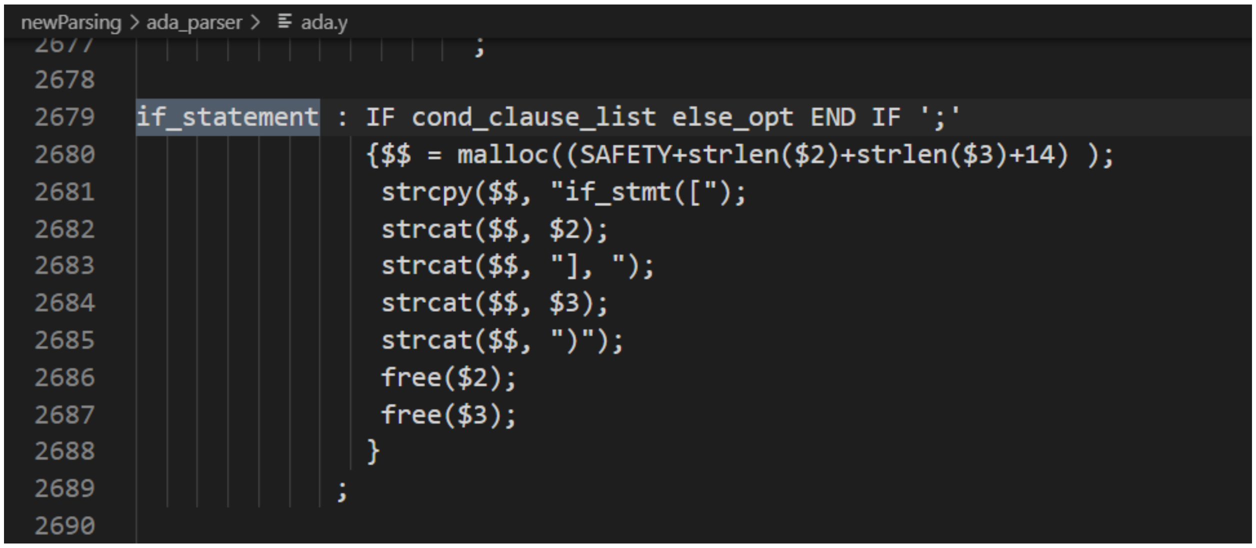
The compilation of the Ada parser went somewhat smoother than the initial step, but there were some hitches encountered here too. Upon selecting the build option the linker was complaining that it was missing files related to the queue. These files were included in the C/C++ include directory as they were for the ourxref compilation. The solution that was achieved was just to place the files that it wanted directly into the project solution and then the compilation successfully completed.

2.2 Parsing

Now with the parser built it was time to run it on some test code, the parser was run on the example code from figure 8 and a supplied example code from Mika relating to dates.

To begin parsing from the command line the following command is entered: mika_ada_parser -M”C:\Mika\bin” -f”C:\GNAT\2010\bin” -gnat05 -d “file to parse” (without extensions). The parser executable was copied into the folder where the target code was present and the above command line entry was tried. Environmental variables had been set for both the Mika\bin and the GNAT\2010\bin but unfortunately there was no bin folder in the Mika project and no way to compile it yet to produce one. A GUI version of Mika had been downloaded to provide examples for how it worked earlier so the \bin folder in that version was referenced instead which was found at “C:\Users\derry\AppData\Roaming\Midoan\Mika\bin” and the parsing completed.

Initially with a very limited understanding of how the parser actually worked, a study of a section of the bison file and how it was parsed in relation to the two sample pieces of code was undertaken. The section in question being the IF statement and how a parser sees that.

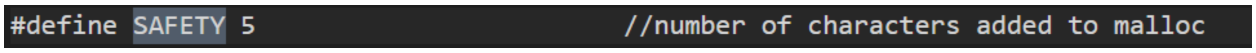


```
2677                        ;
2678
2679 if_statement : IF cond_clause_list else_opt END IF ';'
2680                {$$ = malloc((SAFETY+strlen($2)+strlen($3)+14) );
2681                 strcpy($$, "if_stmt([");
2682                 strcat($$, $2);
2683                 strcat($$, "], ");
2684                 strcat($$, $3);
2685                 strcat($$, ")");
2686                 free($2);
2687                 free($3);
2688                }
2689              ;
2690
```

Figure 6 - IF statement from the ada.y grammar definitions

In figure 20 using bison the rules of how an if statement in Ada is formed is given, the definition on line 2679, if_statement : IF cond_clause_list else_opt END IF ';'. Here it is given a rule that an if statement (if_statement) consists of an if (IF), followed by a list of conditional clauses (cond_clause_list) this can be a list of just one conditional clause or many, followed by an optional else statement (else_opt) this does not need to be present to make a valid if statement, but it is looked for all the same, followed by an end if statement (END IF) and lastly a semicolon (';').

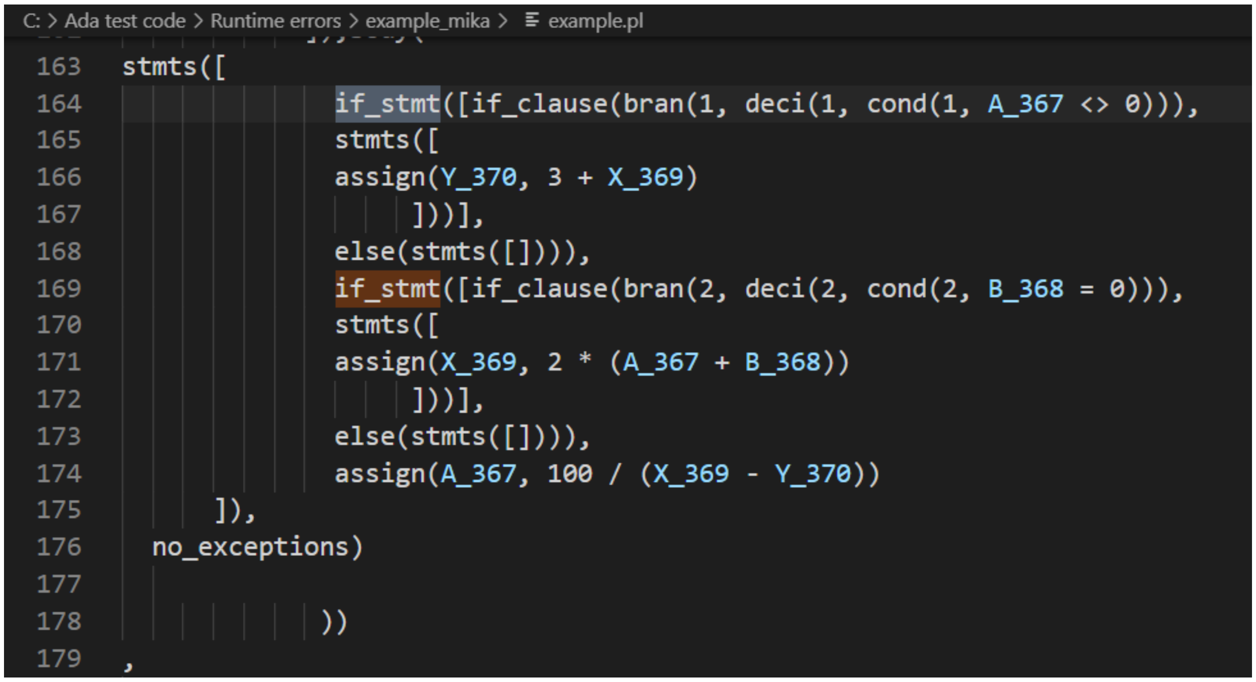
Underneath this definition is how the parser will build up the if statement from the tokenization of the source code. The next line on 2680, {$$ = malloc((SAFETY+strlen($2)+strlen($3)+14)); has a few interesting qualities, firstly the '$$' is the 'if_statement' from the above line and this line of code is an assignment statement, firstly allocating memory for the variable using malloc [30] which is a C language function which allocates memory of a given size and returns a pointer to it. As seen in figure 21 'SAFETY' was defined as 5 within the program, so 5 is added to the length of the second token ($2) and added to the third token ($3), plus 14. Once the memory has been allocated to $$ (if_statement) the code precedes to copy in a string and concatenate further strings to it.



```
#define SAFETY 5                        //number of characters added to malloc
```

Figure 7 - The definition of safety to being the integer 5

Starting with the string "if_stmt([", followed by concatenating the string value of the second token ($2) which was the list of conditional clauses (cond_clause_list), followed by another string "], ", followed by the third token ($3) which was the optional else (else_opt) and finally another string ")". This leads to the building up of the string seen below in figure 8, consisting of lines 164 to 168.


C: > Ada test code > Runtime errors > example_mika > example.pl

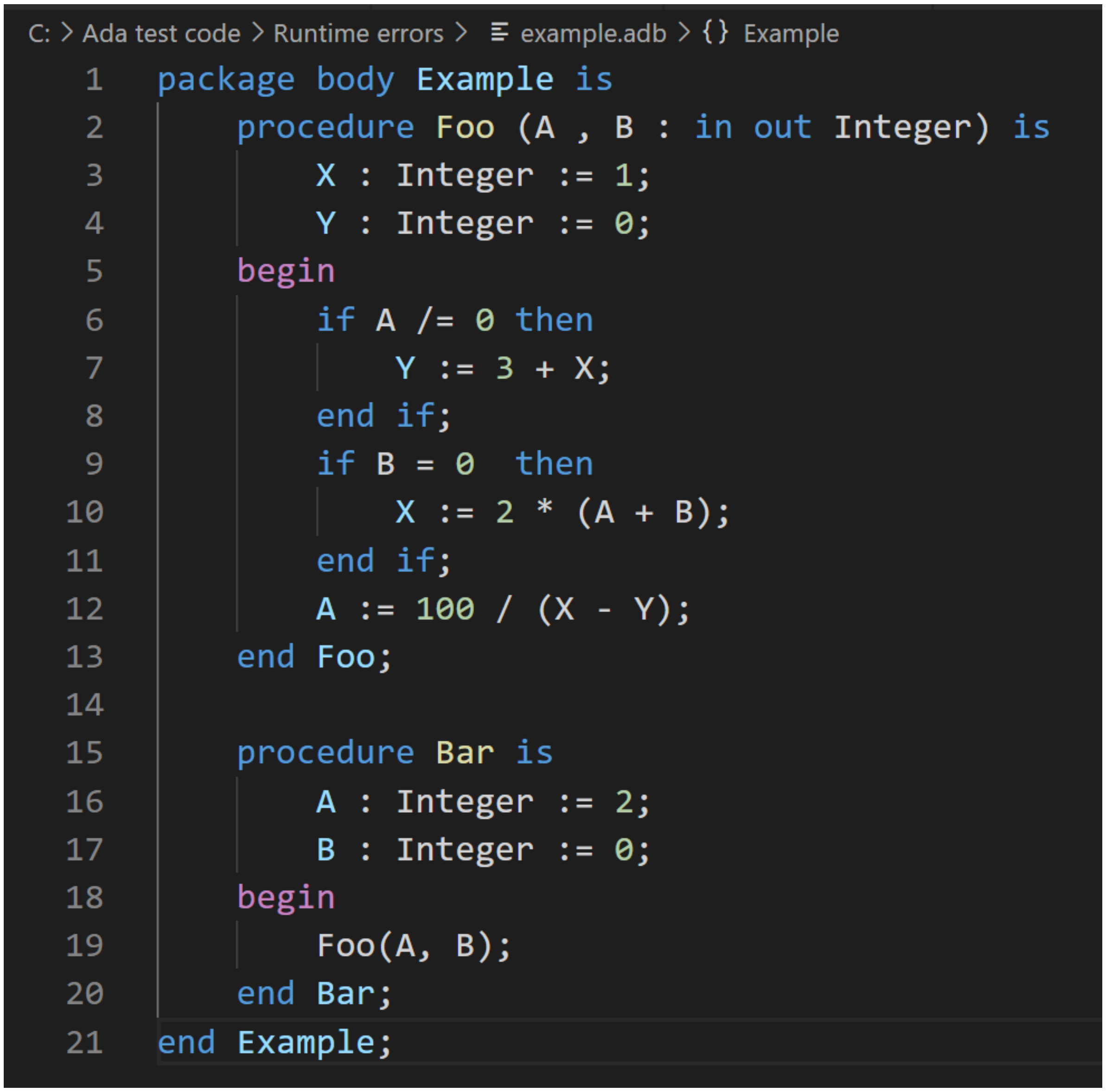
```
stmts([
                if_stmt([if_clause(bran(1, deci(1, cond(1, A_367 <> 0))),
                stmts([
                assign(Y_370, 3 + X_369)
                    ]))],
                else(stmts([]))),
                if_stmt([if_clause(bran(2, deci(2, cond(2, B_368 = 0))),
                stmts([
                assign(X_369, 2 * (A_367 + B_368))
                    ]))],
                else(stmts([]))),
                assign(A_367, 100 / (X_369 - Y_370))
        ]),
 no_exceptions)

                ))
,
```

Figure 8 - How the parser was used to build up a prolog term in example.pl

There is a lot more happening to make up the list of conditional clauses and that is what shall next be looked into.

2.3 Parsing additions

The example program written in ada will be used as the base for the following, this is a piece of code written to make it possible for a division by zero to be possible and should be a good starting point.

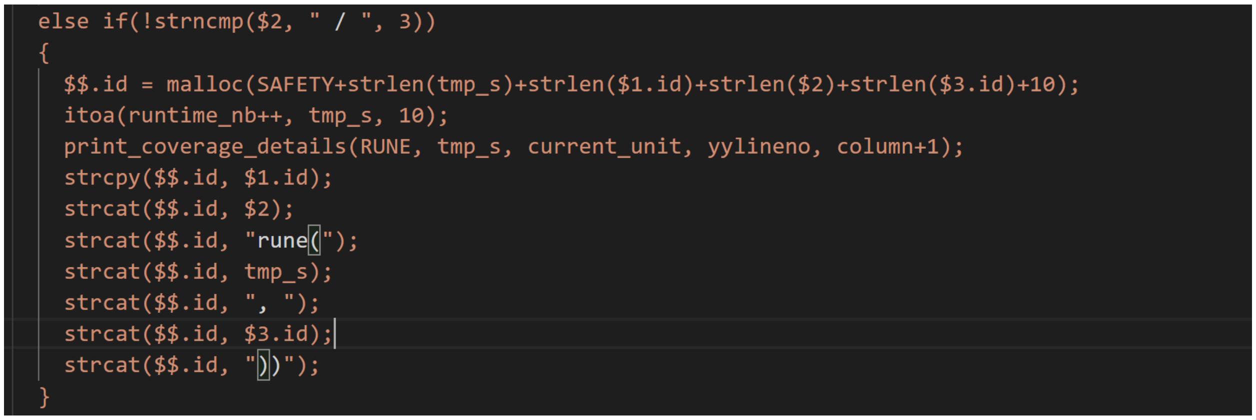


C: > Ada test code > Runtime errors > example.adb > {} Example

```ada
package body Example is
    procedure Foo (A , B : in out Integer) is
        X : Integer := 1;
        Y : Integer := 0;
    begin
        if A /= 0 then
            Y := 3 + X;
        end if;
        if B = 0  then
            X := 2 * (A + B);
        end if;
        A := 100 / (X - Y);
    end Foo;

    procedure Bar is
        A : Integer := 2;
        B : Integer := 0;
    begin
        Foo(A, B);
    end Bar;
end Example;
```

Figure 9 -Example.adb, used to run the as a test program

In order to determine if the divisor is going to be zero, an additional prolog function will need to be constructed to take anything after the division token and determine symbolically if there is a possibility of that being equal to zero.

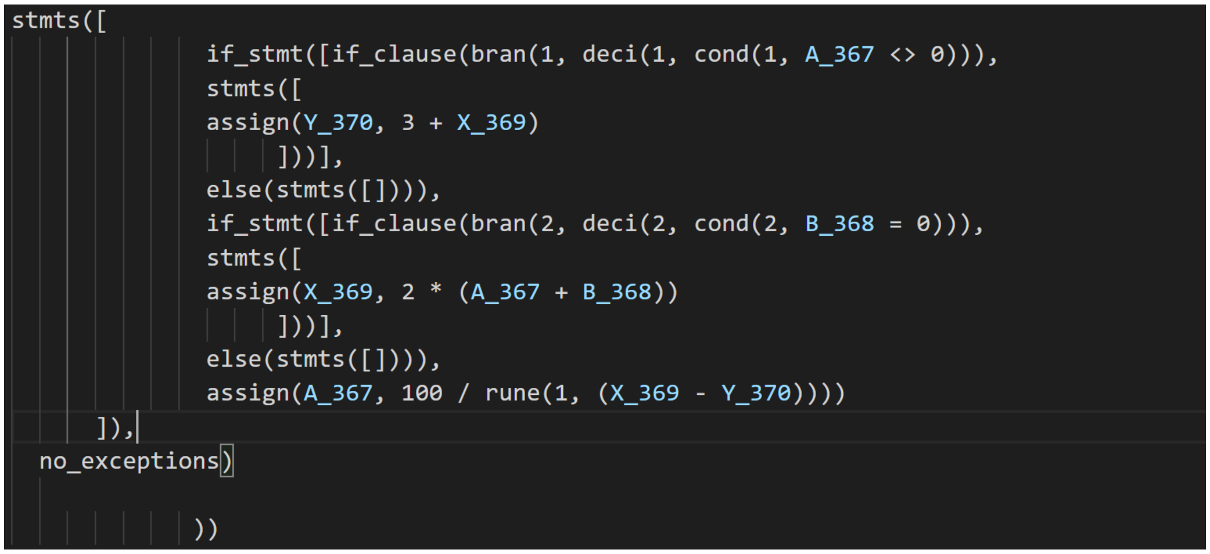


```
else if(!strncmp($2, " / ", 3))
{
  $$.id = malloc(SAFETY+strlen(tmp_s)+strlen($1.id)+strlen($2)+strlen($3.id)+10);
  itoa(runtime_nb++, tmp_s, 10);
  print_coverage_details(RUNE, tmp_s, current_unit, yylineno, column+1);
  strcpy($$.id, $1.id);
  strcat($$.id, $2);
  strcat($$.id, "rune(");
  strcat($$.id, tmp_s);
  strcat($$.id, ", ");
  strcat($$.id, $3.id);
  strcat($$.id, "))");
}
```

Figure 10 - Addition to the ada.y file to construct a new prolog function whenever a division token is encountered

Figure 9 above shows a new else if statement constructed in the ada.y file to explain that whenever a division symbol is encountered it should wrap what follows that token into a new prolog function, called ‘rune’ here for runtime error.

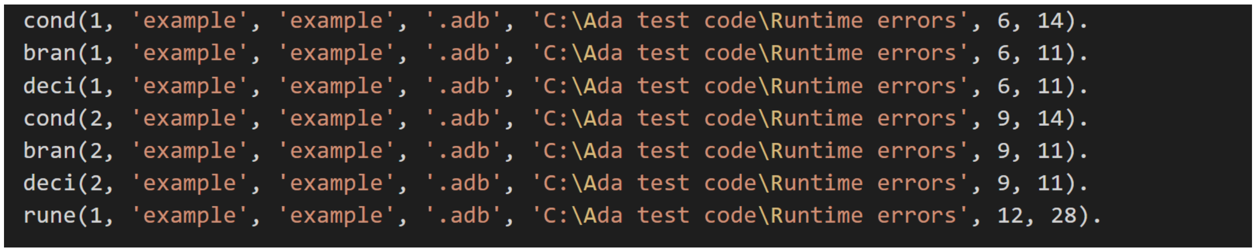
This will then allow prolog and the solver to determine if there is a way for the following tokens to equate to being zero under any circumstance.



```
stmts([
              if_stmt([if_clause(bran(1, deci(1, cond(1, A_367 <> 0))),
              stmts([
              assign(Y_370, 3 + X_369)
                   ]))],
              else(stmts([]))),
              if_stmt([if_clause(bran(2, deci(2, cond(2, B_368 = 0))),
              stmts([
              assign(X_369, 2 * (A_367 + B_368))
                   ]))],
              else(stmts([]))),
              assign(A_367, 100 / rune(1, (X_369 - Y_370))))
      ]),
  no_exceptions)

             ))
```

Figure 11 - New prolog function addition run on example.adb

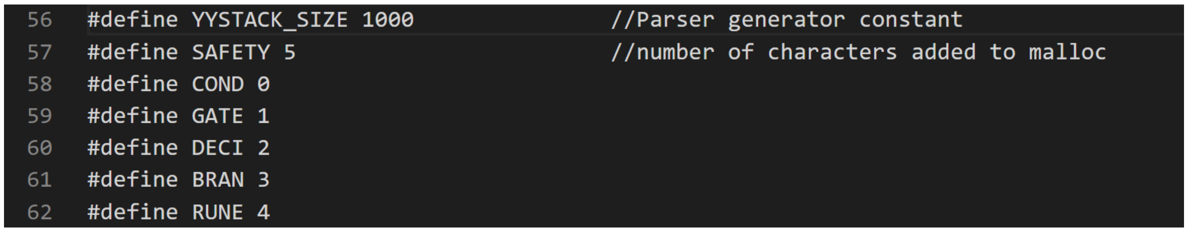
As can be seen in figure 10 above everything after the division token is now past into the rune function. A print coverage detail was also added into the ada.y file and the produced output in the prolog file is as such.



```
cond(1, 'example', 'example', '.adb', 'C:\Ada test code\Runtime errors', 6, 14).
bran(1, 'example', 'example', '.adb', 'C:\Ada test code\Runtime errors', 6, 11).
deci(1, 'example', 'example', '.adb', 'C:\Ada test code\Runtime errors', 6, 11).
cond(2, 'example', 'example', '.adb', 'C:\Ada test code\Runtime errors', 9, 14).
bran(2, 'example', 'example', '.adb', 'C:\Ada test code\Runtime errors', 9, 11).
deci(2, 'example', 'example', '.adb', 'C:\Ada test code\Runtime errors', 9, 11).
rune(1, 'example', 'example', '.adb', 'C:\Ada test code\Runtime errors', 12, 28).
```

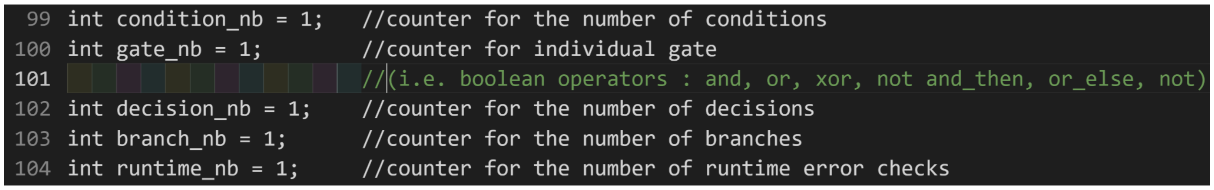
Figure 12 - Coverage details generated in the prolog file

Other minor additions were also added to the ada.y file such as the definition of RUNE as 4, instantiating the an integer variable named runtime_nb as 1, used to track the number of times the RUNE function has been called and a new case added to the switch statement to handle the the additional rune function also.



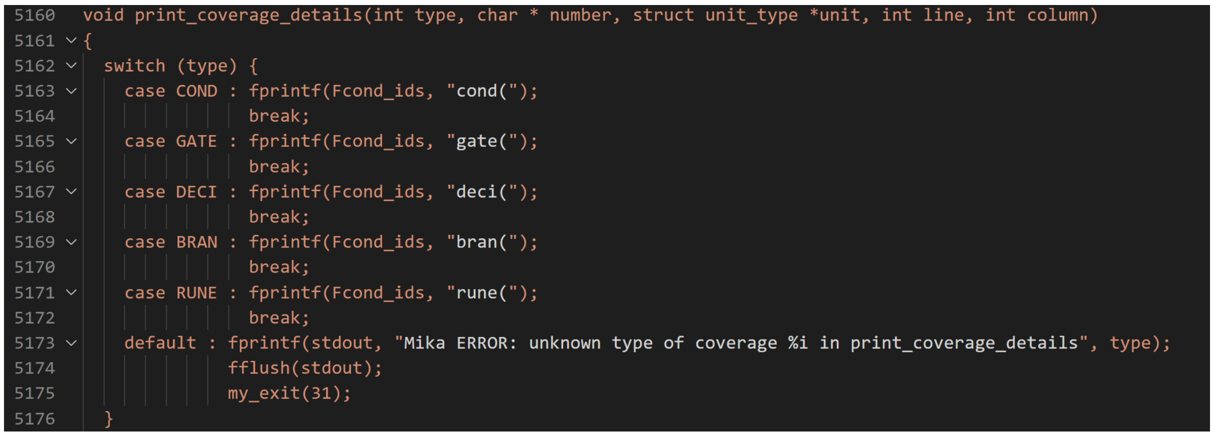
```c
#define YYSTACK_SIZE 1000                   //Parser generator constant
#define SAFETY 5                            //number of characters added to malloc
#define COND 0
#define GATE 1
#define DECI 2
#define BRAN 3
#define RUNE 4
```

Figure 13 - Definition of RUNE



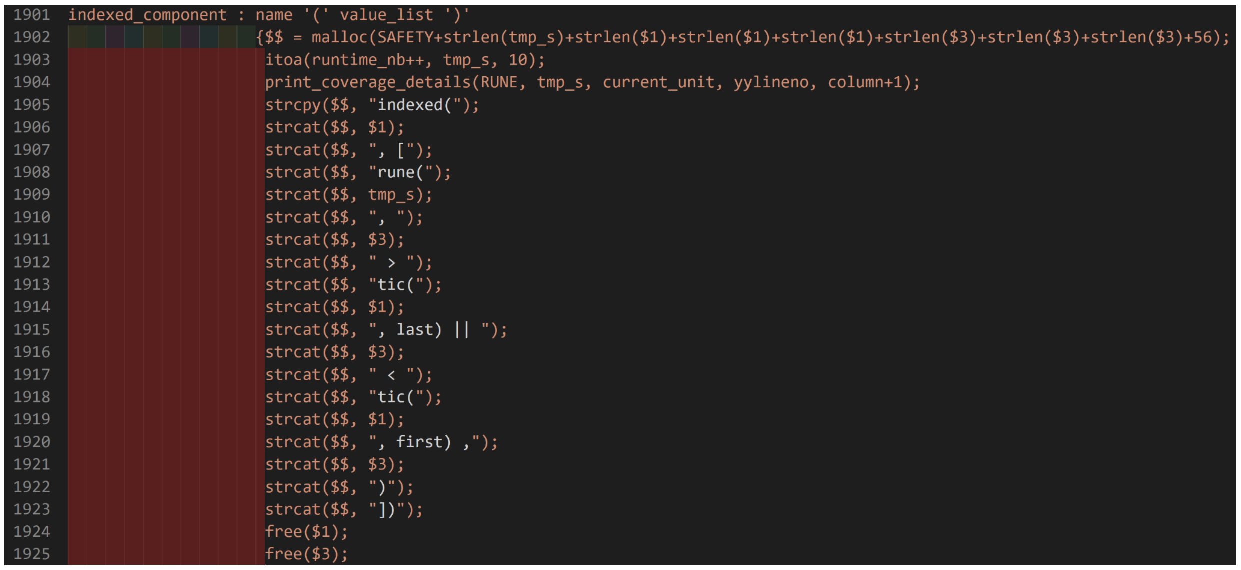
```c
int condition_nb = 1;   //counter for the number of conditions
int gate_nb = 1;        //counter for individual gate
                        //(i.e. boolean operators : and, or, xor, not and_then, or_else, not)
int decision_nb = 1;    //counter for the number of decisions
int branch_nb = 1;      //counter for the number of branches
int runtime_nb = 1;     //counter for the number of runtime error checks
```

Figure 14 - Instantiation of runtime_nb



```c
void print_coverage_details(int type, char * number, struct unit_type *unit, int line, int column)
{
  switch (type) {
    case COND : fprintf(Fcond_ids, "cond(");
                break;
    case GATE : fprintf(Fcond_ids, "gate(");
                break;
    case DECI : fprintf(Fcond_ids, "deci(");
                break;
    case BRAN : fprintf(Fcond_ids, "bran(");
                break;
    case RUNE : fprintf(Fcond_ids, "rune(");
                break;
    default : fprintf(stdout, "Mika ERROR: unknown type of coverage %i in print_coverage_details", type);
              fflush(stdout);
              my_exit(31);
  }
```

Figure 15 - Print coverage details additional switch case



```
indexed_component : name '(' value_list ')'
                    {$$ = malloc(SAFETY+strlen(tmp_s)+strlen($1)+strlen($1)+strlen($1)+strlen($3)+strlen($3)+strlen($3)+56);
                     itoa(runtime_nb++, tmp_s, 10);
                     print_coverage_details(RUNE, tmp_s, current_unit, yylineno, column+1);
                     strcpy($$, "indexed(");
                     strcat($$, $1);
                     strcat($$, ", [");
                     strcat($$, "rune(");
                     strcat($$, tmp_s);
                     strcat($$, ", ");
                     strcat($$, $3);
                     strcat($$, " > ");
                     strcat($$, "tic(");
                     strcat($$, $1);
                     strcat($$, ", last) || ");
                     strcat($$, $3);
                     strcat($$, " < ");
                     strcat($$, "tic(");
                     strcat($$, $1);
                     strcat($$, ", first) ,");
                     strcat($$, $3);
                     strcat($$, ")");
                     strcat($$, "])");
                     free($1);
                     free($3);
```

Figure 16 - additions to indexed_component to construct the correct checks within the generated prolog file

Further additions were made to the ada.y parser to take into account the indexed element used when an array is being indexed with a value. This addition took the same form as the division by zero additions, it just increased in complexity.

The usage of the tic() here is used to make use of the Ada element'First and element'Last that returns the value of the first and last index value of what is being referenced. This would construct a prolog relationship to determine if the supplied value fell within the range of the index of the array. At this stage it became apparent that the parser was not going to be a viable option for the continuation of this approach, as the parser is not smart enough to be able to tell the types of the elements it is dealing with. For instance an indexed component could be any of the following: an array access, a function or procedure call, a type conversion, or a subtype indication with index constraint. As the purpose of the inserted additions focused solely on array indexing, the additions that would be inserted upon encountering any of the other indexed elements would either be unnecessary or could even cause a crash.

3. Extension Idea

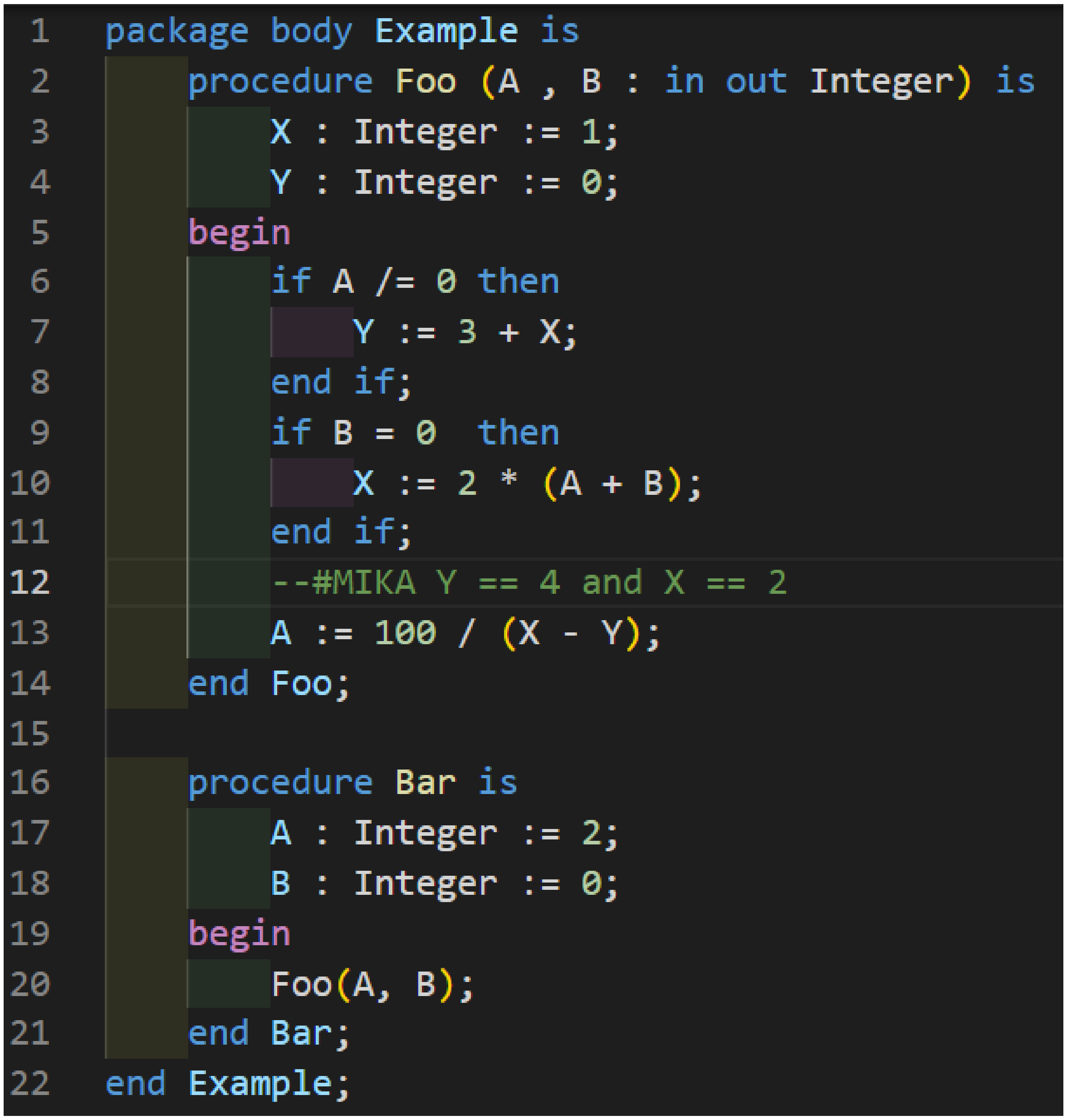
As discussed above the idea for the extension would be to have the developer insert an annotation into the code at a specified line asking under what conditions certain variables could have the provided values. The extension would then run the saved code through the parser where there would have to be adjustments made to handle these comments and add an additional step to the generated prolog file. Once the file has been parsed then the test input generation would be able to provide inputs where available to match the supplied annotation.

There are a few hurdles to consider here, mainly that to run the parser there are certain paths that need to be provided, the most simple option may be to have the developer supply these paths in other annotations at the top of the code and then they could be pulled out and run in the command line once the parser is running.

Upon further research there is an extension settings page available [54] where these paths can be supplied directly and then the extension can pull from these values upon calling Mika.

The parser (mika_ada_parser.exe) and the test input generator (mika_ada_generator.exe) also need to be present in the current path where the command line is being run from, so these would have to be copied over to the working directory prior to running any commands.

The Mika parser and generator can also be referenced relatively and as such no need for copying files is needed here.



```ada
package body Example is
    procedure Foo (A , B : in out Integer) is
        X : Integer := 1;
        Y : Integer := 0;
    begin
        if A /= 0 then
            Y := 3 + X;
        end if;
        if B = 0  then
            X := 2 * (A + B);
        end if;
        --#MIKA Y == 4 and X == 2
        A := 100 / (X - Y);
    end Foo;

    procedure Bar is
        A : Integer := 2;
        B : Integer := 0;
    begin
        Foo(A, B);
    end Bar;
end Example;
```

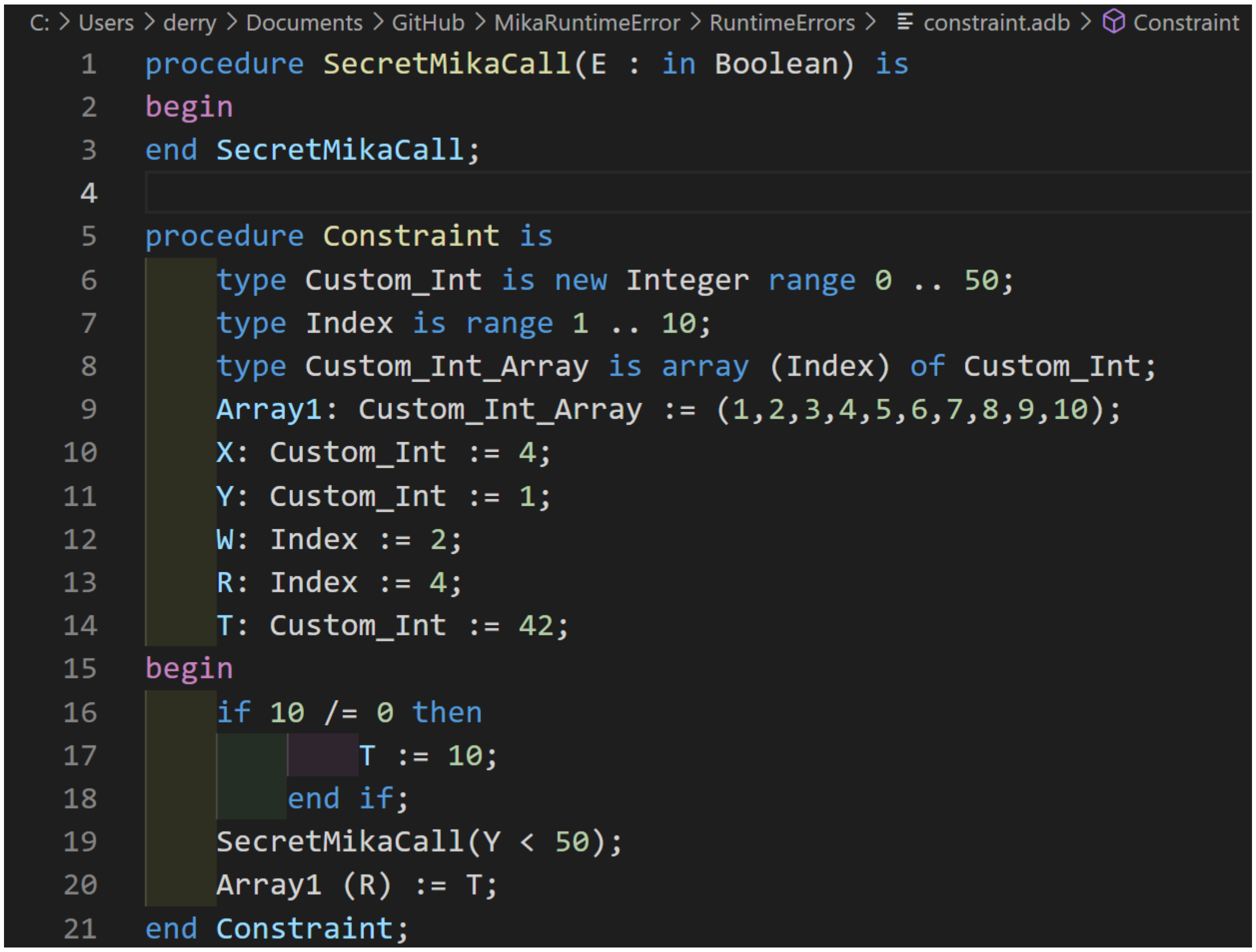
Figure 17 - Example Ada code used to demonstrate the special --#Mika comment to be used

In order to execute the proposed idea, firstly an addition to the lexical analyzer needs to be made. A token to represent the comment “--#MIKA.*” needs to be written. Comments are usually ignored during the lexical analyzer phase but now a new condition or regex needs to be implemented that will add what follows the start of the comment as valid code to be worked upon in the parser.

The extension can scan the source code for the presence of the special comment “--#MIKA*” and then pull out the string following the “MIKA” in the comment. In this way the boolean condition supplied by the developer can be inserted into a procedure call that will then be analyzed by Mika for its validity and the variable values that would lead it to be true.

Another idea would be to have the developer highlight the line they are interested in and supply valid Ada code in the form of a boolean expression in relation to the variables they are looking to have test input generated for to reach this line of code and have those variables match the provided condition. Using the example from above, the developer highlights line 12 and provides Y = 4 and X = 2.

The extension would then make a copy of the code and insert a dummy procedure that would take in a boolean as an argument and also insert a call of this procedure at the specified line with the provided boolean condition inserted as its parameter. By taking this route the cross referencer will know that the variables that are being referred to are the ones within the scope of the code block, the comment route would have left the specific variable very ambiguous. This way will also preserve the source code and no edits would have to be carried out on that as the copy of the code can be deleted once the test inputs have been generated. Some of the complications that arise from this choice will be that the line numbering from the source code and the copied code with a procedure call and the procedure itself will not match up. An example is shown below of how this might look within a previously used example code, with the dummy procedure added from line 1-3 and the procedure itself being called on line 19 within the constraint procedure.



C: > Users > derry > Documents > GitHub > MikaRuntimeError > RuntimeErrors > constraint.adb > Constraint

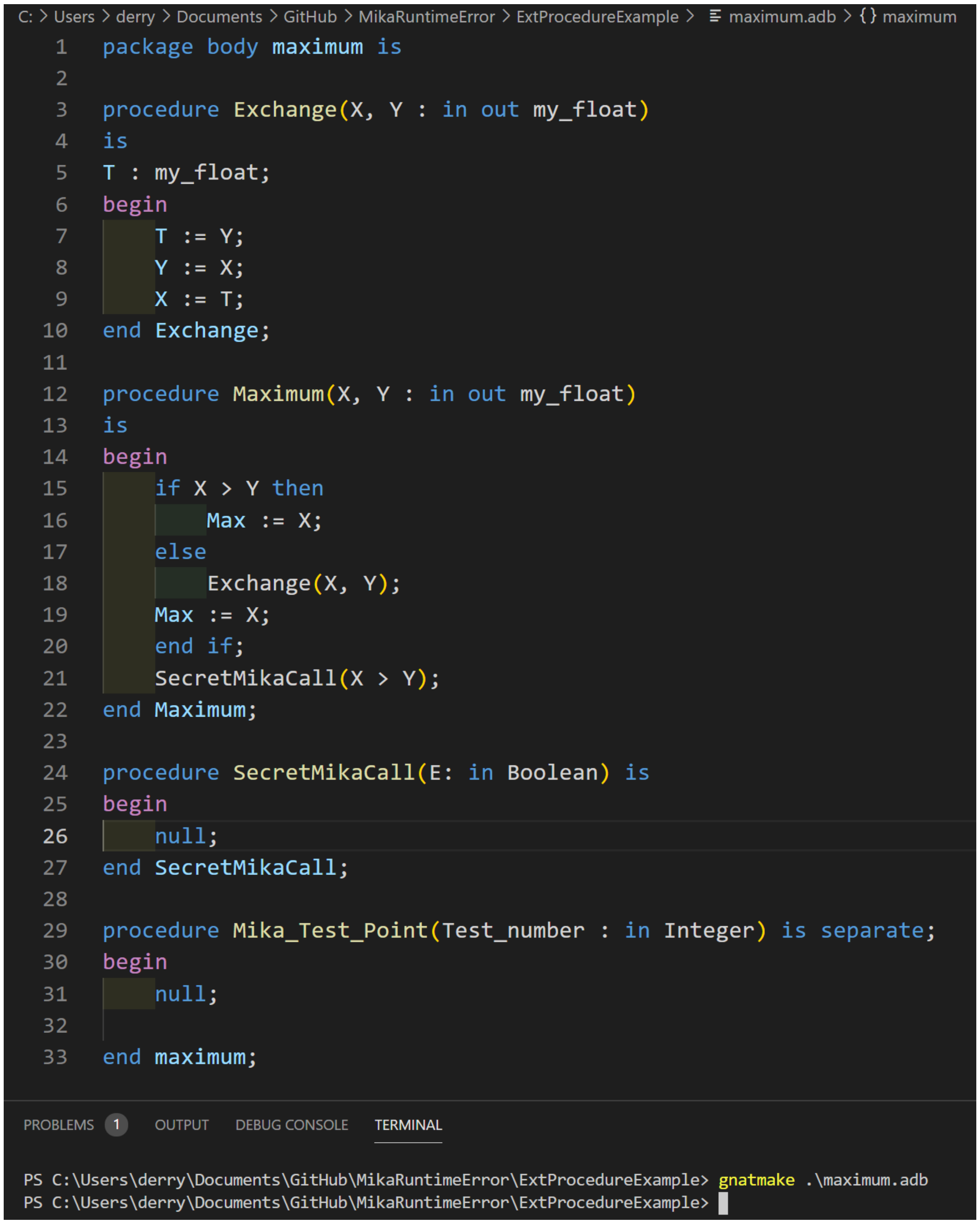
```
procedure SecretMikaCall(E : in Boolean) is
begin
end SecretMikaCall;

procedure Constraint is
    type Custom_Int is new Integer range 0 .. 50;
    type Index is range 1 .. 10;
    type Custom_Int_Array is array (Index) of Custom_Int;
    Array1: Custom_Int_Array := (1,2,3,4,5,6,7,8,9,10);
    X: Custom_Int := 4;
    Y: Custom_Int := 1;
    W: Index := 2;
    R: Index := 4;
    T: Custom_Int := 42;
begin
    if 10 /= 0 then
            T := 10;
        end if;
    SecretMikaCall(Y < 50);
    Array1 (R) := T;
end Constraint;
```

Figure 18 - Ada code to show an example of the SecretMikaCall procedure definition and call within a program

As a proof of concept the manual addition of this procedure will be inserted into the packages that come as example code with mika. Instead of the procedure being inserted at the top of the package it will be inserted at the end of the package instead. This will be to account for a variable number of imports that a package might have and also decrease the calculations needed to maintain the correct line numberings with the original file.

Both the maxim.adb and array_date.adb compiled with the addition of the dummy procedure. This was not without issues,as it required an addition to the .ads file also and it was required to be above the Mika_Test_Point procedure.This seems to be due to this procedure being housed in a separate file as specified by the “is separate” in the procedure declaration.

This could cause some issues if using separate procedures is common outside of these particular examples as it would require additional steps to determine if a separate procedure is present in the code.

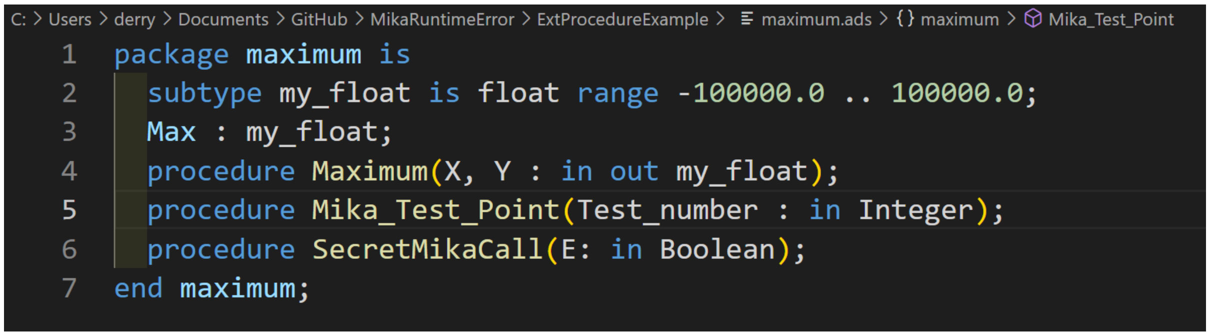


C: > Users > derry > Documents > GitHub > MikaRuntimeError > ExtProcedureExample > maximum.adb > {} maximum

```
package body maximum is

procedure Exchange(X, Y : in out my_float)
is
T : my_float;
begin
    T := Y;
    Y := X;
    X := T;
end Exchange;

procedure Maximum(X, Y : in out my_float)
is
begin
    if X > Y then
        Max := X;
    else
        Exchange(X, Y);
    Max := X;
    end if;
    SecretMikaCall(X > Y);
end Maximum;

procedure SecretMikaCall(E: in Boolean) is
begin
    null;
end SecretMikaCall;

procedure Mika_Test_Point(Test_number : in Integer) is separate;
begin
    null;

end maximum;
```

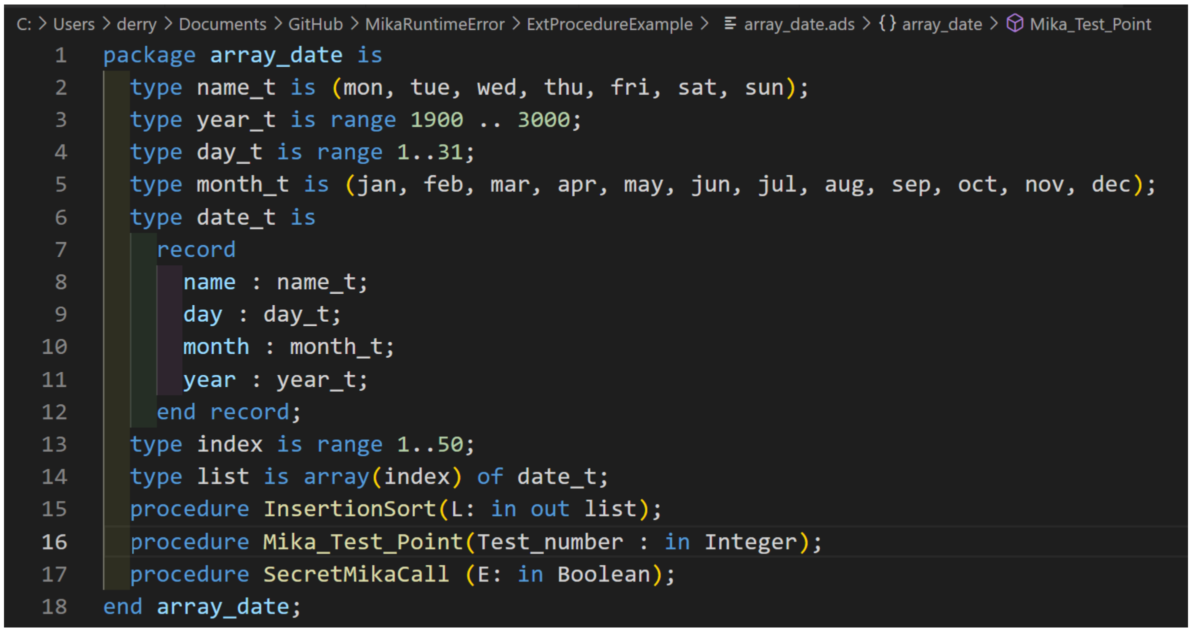
```
PS C:\Users\derry\Documents\GitHub\MikaRuntimeError\ExtProcedureExample> gnatmake .\maximum.adb
PS C:\Users\derry\Documents\GitHub\MikaRuntimeError\ExtProcedureExample>
```

Figure 19 - A test to see if the procedure definition could come after the call in the flow of a package



```
C: > Users > derry > Documents > GitHub > MikaRuntimeError > ExtProcedureExample > maximum.ads > {} maximum > Mika_Test_Point
package maximum is
   subtype my_float is float range -100000.0 .. 100000.0;
   Max : my_float;
   procedure Maximum(X, Y : in out my_float);
   procedure Mika_Test_Point(Test_number : in Integer);
   procedure SecretMikaCall(E: in Boolean);
end maximum;
```

Figure 20 - figure 19 layout only works with an addition to the files .ads file also

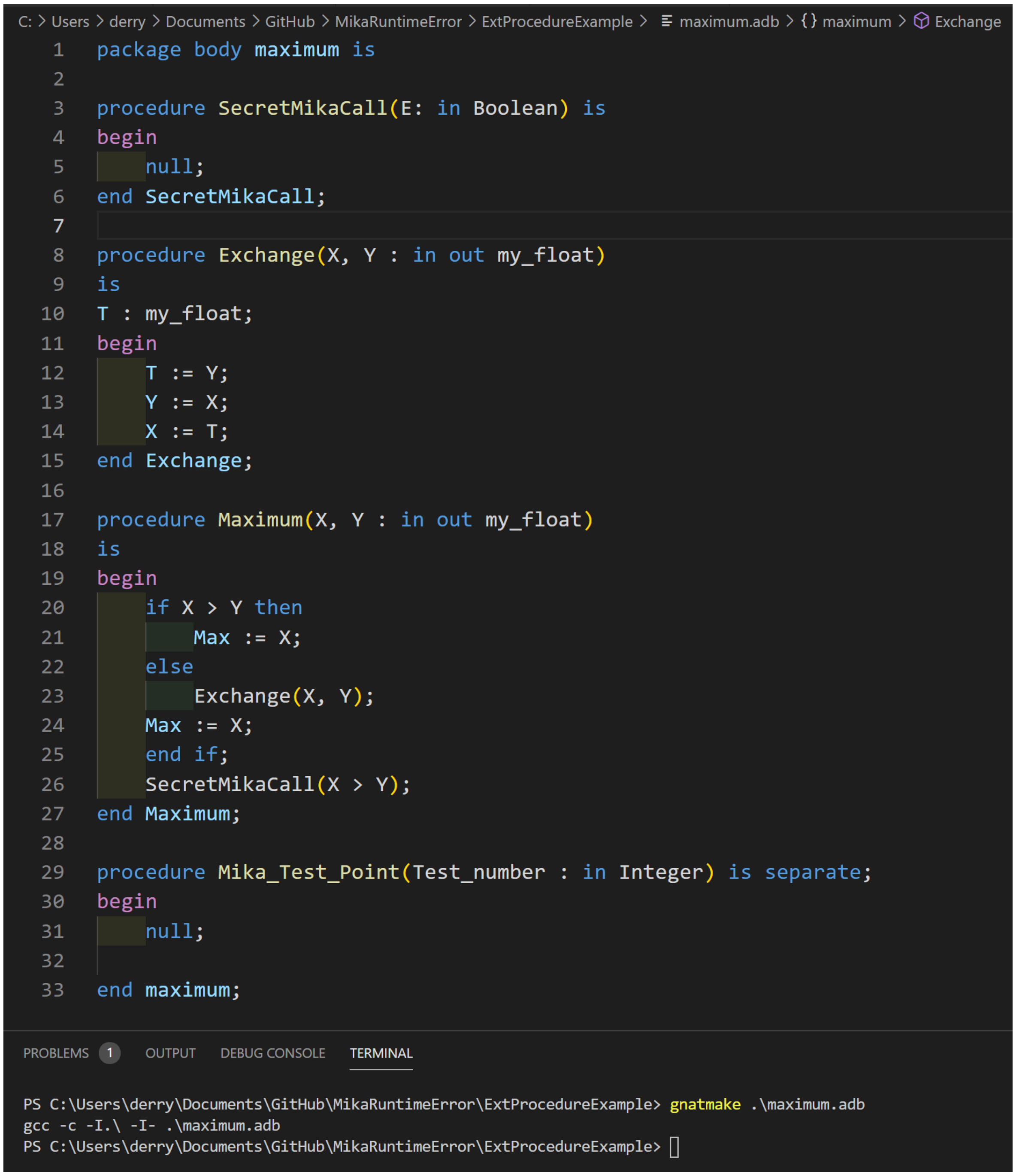


```
C: > Users > derry > Documents > GitHub > MikaRuntimeError > ExtProcedureExample > array_date.ads > {} array_date > Mika_Test_Point
package array_date is
   type name_t is (mon, tue, wed, thu, fri, sat, sun);
   type year_t is range 1900 .. 3000;
   type day_t is range 1..31;
   type month_t is (jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec);
   type date_t is
      record
         name : name_t;
         day : day_t;
         month : month_t;
         year : year_t;
      end record;
   type index is range 1..50;
   type list is array(index) of date_t;
   procedure InsertionSort(L: in out list);
   procedure Mika_Test_Point(Test_number : in Integer);
   procedure SecretMikaCall (E: in Boolean);
end array_date;
```

Figure 21 - another test of inserting procedure definition into an .ads file, it was determined this was unnecessarily complicated

The addition of the procedure to the .ads file was necessary as the procedure was being called before its declaration in the code and at the time of the call the procedure is unknown if it is not supplied in the .ads file. A test of this with adding the dummy procedure call to the top of the package and removing it from the .ads file will confirm this.

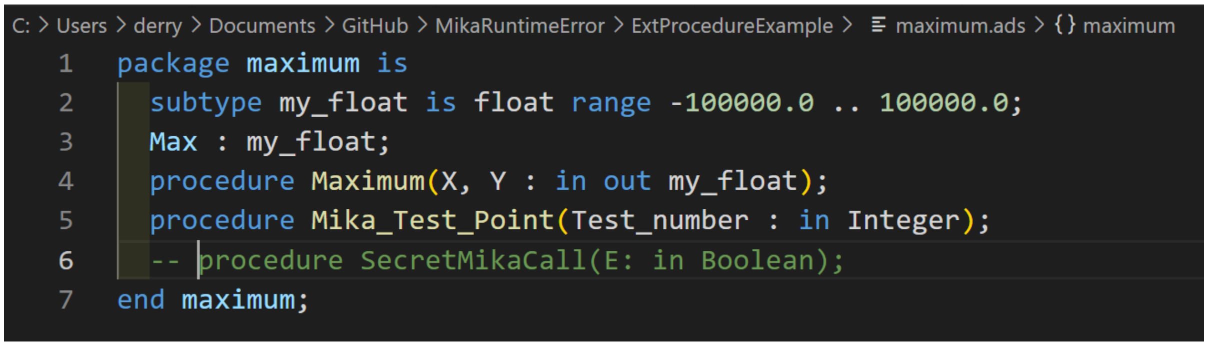
As can be seen in the below images, having the dummy procedure at the top of the package only requires for the .adb file to have an addition made to it. Going forward the best practice of where the insertion should take place will be determined by some practical experimentation.


C: > Users > derry > Documents > GitHub > MikaRuntimeError > ExtProcedureExample > maximum.adb > { } maximum > Exchange
PROBLEMS 1
OUTPUT
DEBUG CONSOLE
TERMINAL

```ada
package body maximum is

procedure SecretMikaCall(E: in Boolean) is
begin
    null;
end SecretMikaCall;

procedure Exchange(X, Y : in out my_float)
is
T : my_float;
begin
    T := Y;
    Y := X;
    X := T;
end Exchange;

procedure Maximum(X, Y : in out my_float)
is
begin
    if X > Y then
        Max := X;
    else
        Exchange(X, Y);
    Max := X;
    end if;
    SecretMikaCall(X > Y);
end Maximum;

procedure Mika_Test_Point(Test_number : in Integer) is separate;
begin
    null;

end maximum;
```

```
PS C:\Users\derry\Documents\GitHub\MikaRuntimeError\ExtProcedureExample> gnatmake .\maximum.adb
gcc -c -I.\ -I- .\maximum.adb
PS C:\Users\derry\Documents\GitHub\MikaRuntimeError\ExtProcedureExample>
```

Figure 22 - By defining the SecretMikaCall procedure at the start of the package, no additions to the .ads file were necessary


C: > Users > derry > Documents > GitHub > MikaRuntimeError > ExtProcedureExample > maximum.ads > {} maximum

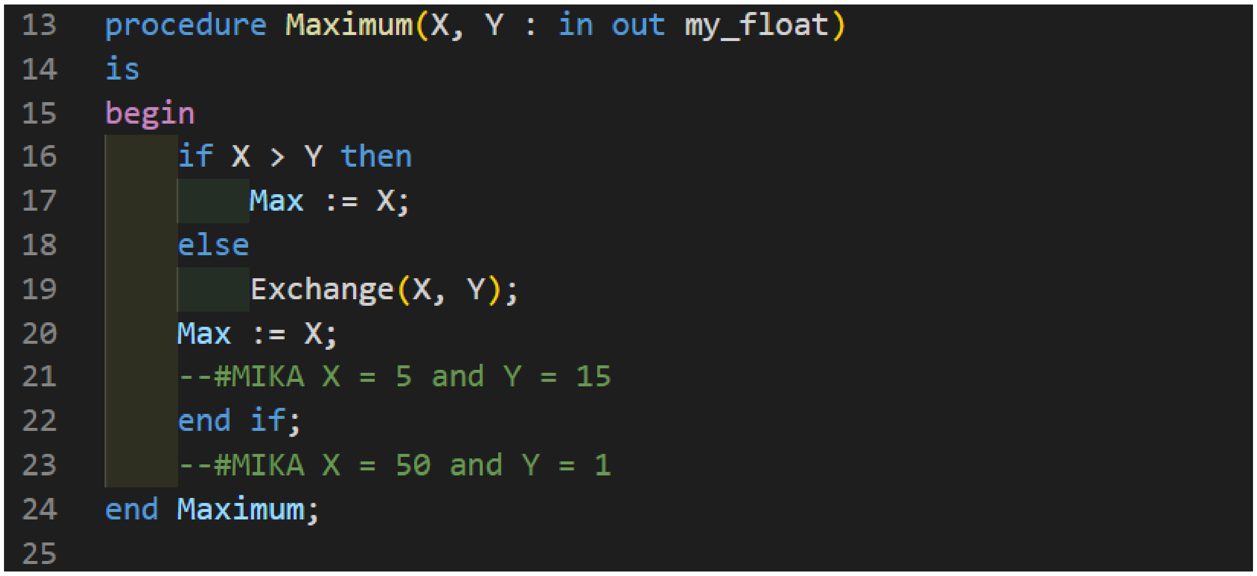
```ada
package maximum is
   subtype my_float is float range -100000.0 .. 100000.0;
   Max : my_float;
   procedure Maximum(X, Y : in out my_float);
   procedure Mika_Test_Point(Test_number : in Integer);
   -- procedure SecretMikaCall(E: in Boolean);
end maximum;
```

Figure 23 - Removal of the SecretMikaCall from the .ads file

Another problem that was encountered in the process of making a copy of the files being worked on was that a change in the name of the file also required a change in the name of the package within the file at both the definition of the package and at its end. This also spilled over into the packages .ads file which would also need to have alterations made to it in order to compile the file. Whereas if a new folder was created and all the files in the current directory were copied over, alterations could be made to the file in question while maintaining the original integrity and avoiding the need for messy renaming.

This requires a bit of refactoring of what had already been done, but will hopefully pay off in the future.

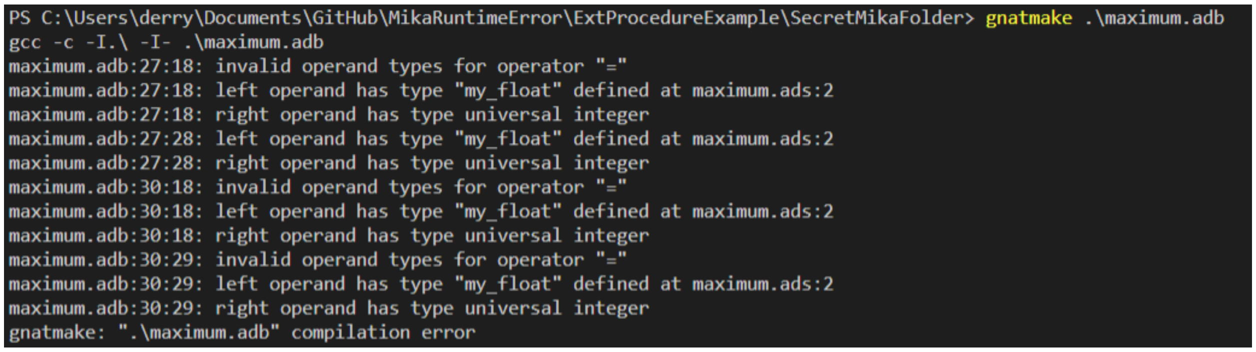
At present the extension has two commands. One command is for the developer to enter a comment into the code at the line they wish to have a particular test carried out.



```
procedure Maximum(X, Y : in out my_float)
is
begin
    if X > Y then
        Max := X;
    else
        Exchange(X, Y);
    Max := X;
    --#MIKA X = 5 and Y = 15
    end if;
    --#MIKA X = 50 and Y = 1
end Maximum;

```

Figure 24 - Example of the Mika comments within code and the structure that they take

The example above is not correct though as X and Y are both of my_float type and this would not compile as Gnat sees a comparison between different types as a compilation error.



```
PS C:\Users\derry\Documents\GitHub\MikaRuntimeError\ExtProcedureExample\SecretMikaFolder> gnatmake .\maximum.adb
gcc -c -I.\ -I- .\maximum.adb
maximum.adb:27:18: invalid operand types for operator "="
maximum.adb:27:18: left operand has type "my_float" defined at maximum.ads:2
maximum.adb:27:18: right operand has type universal integer
maximum.adb:27:28: left operand has type "my_float" defined at maximum.ads:2
maximum.adb:27:28: right operand has type universal integer
maximum.adb:30:18: invalid operand types for operator "="
maximum.adb:30:18: left operand has type "my_float" defined at maximum.ads:2
maximum.adb:30:18: right operand has type universal integer
maximum.adb:30:29: invalid operand types for operator "="
maximum.adb:30:29: left operand has type "my_float" defined at maximum.ads:2
maximum.adb:30:29: right operand has type universal integer
gnatmake: ".\maximum.adb" compilation error
```

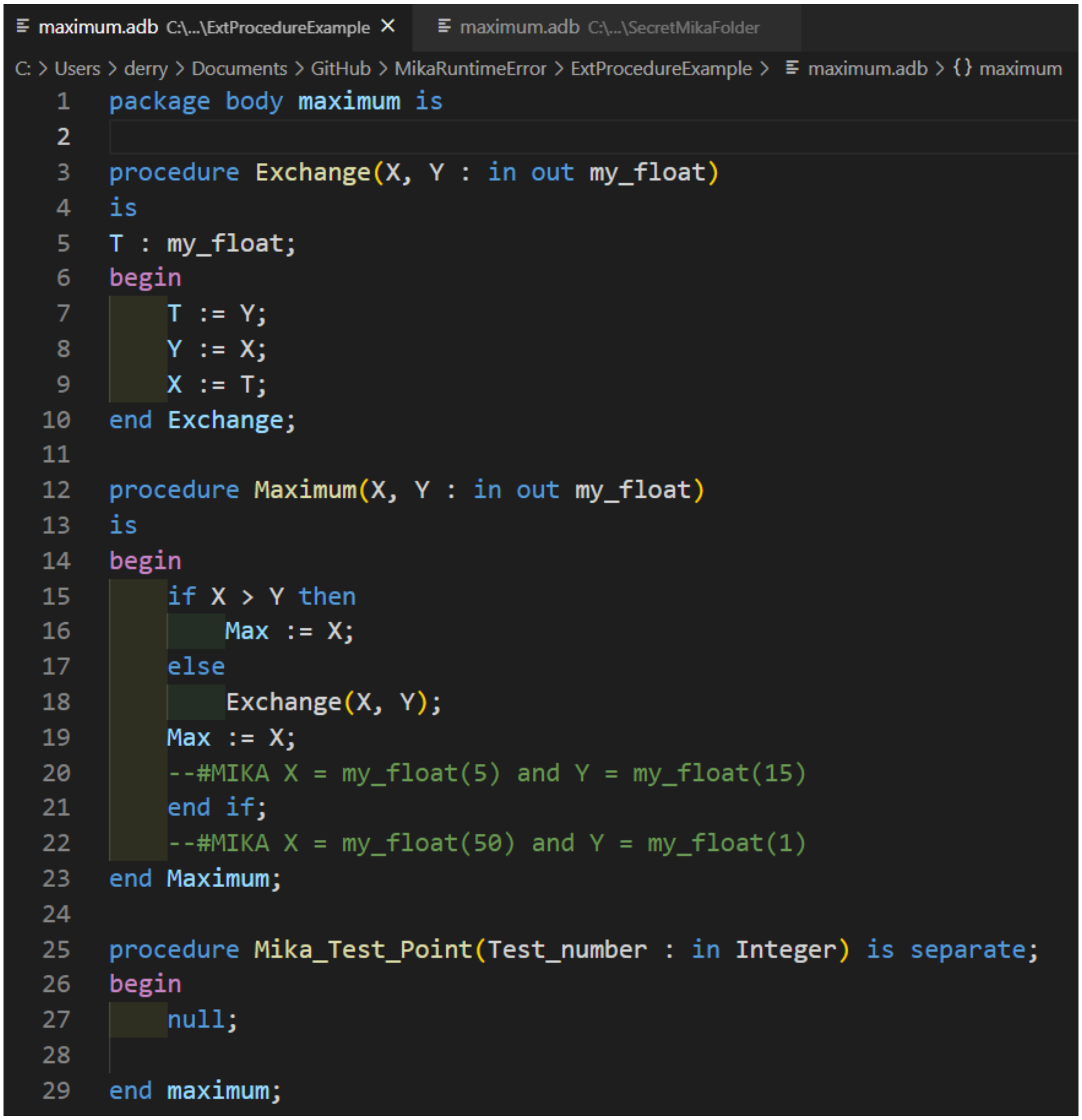
Figure 25 - Errors highlighting the fact that the developer must supply correct syntax and typing for the boolean condition. Here Integers were entered while X & Y were of type my_float

It will be up to the developer to provide valid Ada code in the comment provided. If the provided comment is not valid Ada code the compilation of the generated file will fail and an error message of "Mika failed to generate test inputs, Please check syntax of supplied comment condition" will be displayed to the user.

The second command is run after comments have been entered into the code. This will make a new folder in the directory where the original Ada file was located called “secretMikaFolder”. It also copies in the files from that directory to account for the .ads file and any other dependencies.

It then inserts the procedure calls with the supplied parameters into the line numbers provided and also the definition of the SecretMikaCall to the start of the package.

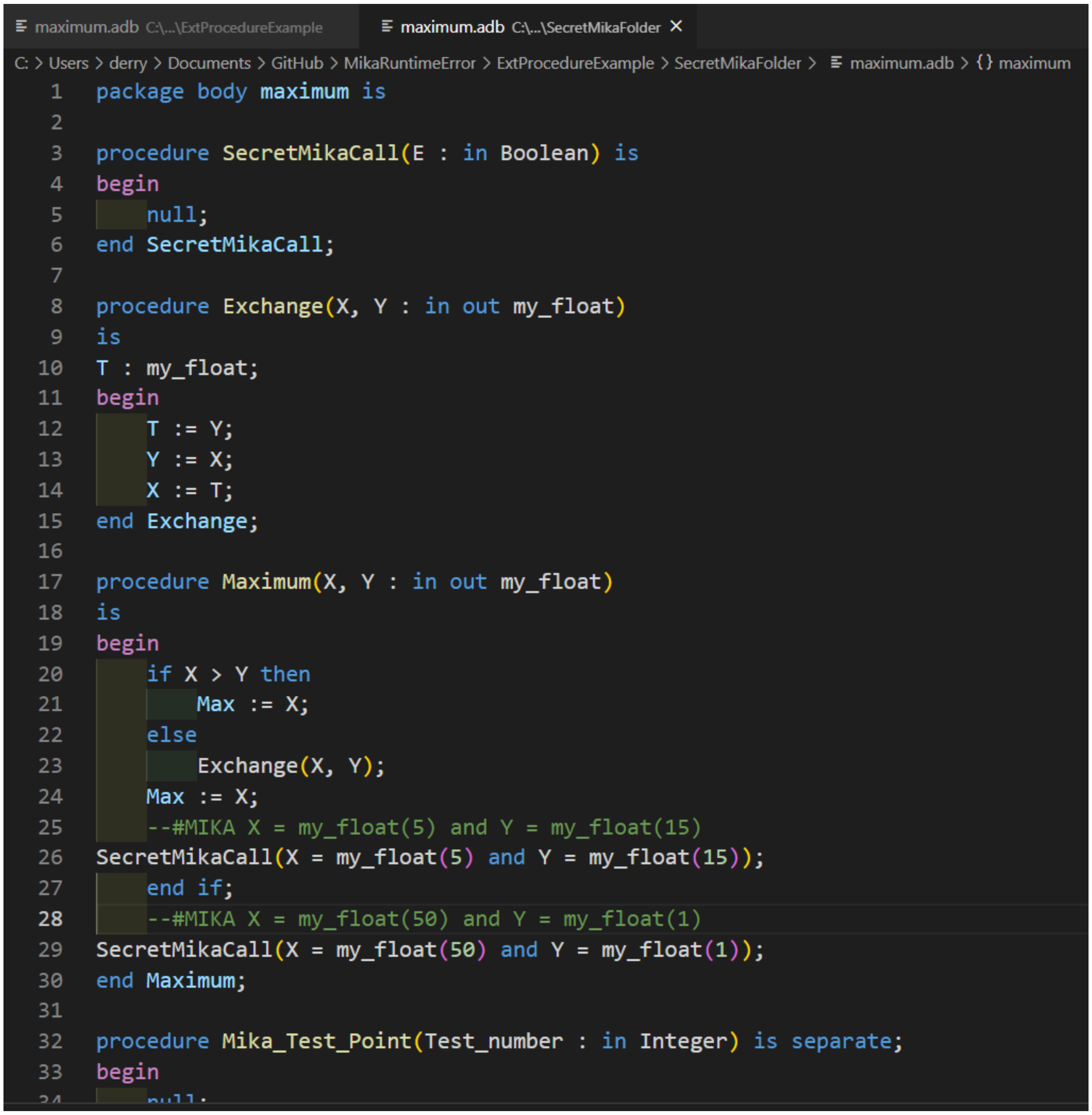
Below is an image of the original file with the developer comments provided.


maximum.adb C:\...\ExtProcedureExample
maximum.adb C:\...\SecretMikaFolder
C: > Users > derry > Documents > GitHub > MikaRuntimeError > ExtProcedureExample > maximum.adb > {} maximum

```ada
package body maximum is

procedure Exchange(X, Y : in out my_float)
is
T : my_float;
begin
    T := Y;
    Y := X;
    X := T;
end Exchange;

procedure Maximum(X, Y : in out my_float)
is
begin
    if X > Y then
        Max := X;
    else
        Exchange(X, Y);
    Max := X;
    --#MIKA X = my_float(5) and Y = my_float(15)
    end if;
    --#MIKA X = my_float(50) and Y = my_float(1)
end Maximum;

procedure Mika_Test_Point(Test_number : in Integer) is separate;
begin
    null;

end maximum;
```

Figure 26 - comments entered while respecting the correct types used within the program

Followed by the copied file in the new folder with the procedure definition and calls programmatically entered.


maximum.adb C:\...\ExtProcedureExample
maximum.adb C:\...\SecretMikaFolder
C: > Users > derry > Documents > GitHub > MikaRuntimeError > ExtProcedureExample > SecretMikaFolder > maximum.adb > {} maximum

```
package body maximum is

procedure SecretMikaCall(E : in Boolean) is
begin
    null;
end SecretMikaCall;

procedure Exchange(X, Y : in out my_float)
is
T : my_float;
begin
    T := Y;
    Y := X;
    X := T;
end Exchange;

procedure Maximum(X, Y : in out my_float)
is
begin
    if X > Y then
        Max := X;
    else
        Exchange(X, Y);
    Max := X;
    --#MIKA X = my_float(5) and Y = my_float(15)
SecretMikaCall(X = my_float(5) and Y = my_float(15));
    end if;
    --#MIKA X = my_float(50) and Y = my_float(1)
SecretMikaCall(X = my_float(50) and Y = my_float(1));
end Maximum;

procedure Mika_Test_Point(Test_number : in Integer) is separate;
begin
```

Figure 27 - Example of the programmatically entered Procedure declaration and calls for each comment

The tab spacing is off but gnat has no problem understanding and compiling this code and as the copied file will be deleted after test inputs are generated this shouldn't be an issue.

The next issues to tackle are the calling of the mika_ada_parser and the mika_ada_generator from the terminal within visual studio code and handling the deletion of the new folder containing the copied files after the test inputs have been generated.

Some consideration has to be taken of line numbering also, as the insertion of the procedure definition and calls will now not match to the line numbering in the source file.

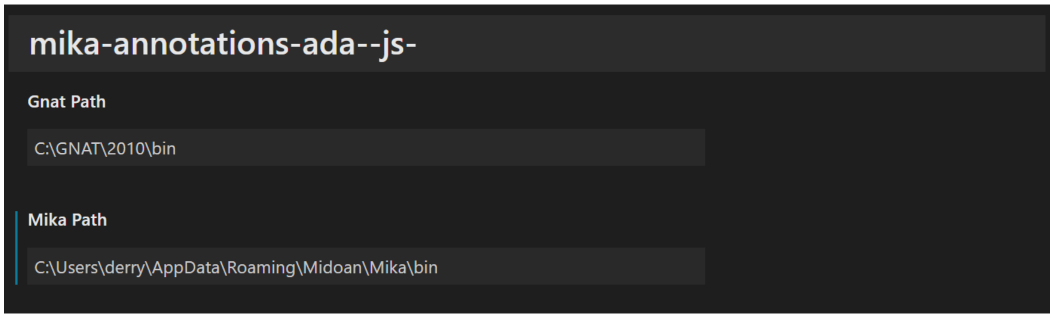
Also there will be a new case in the parser looking for this SecretMikaCall() along the same lines as what was written for the runtime exceptions, the rune().

Moving on, it is also necessary to consider the programmatic implementation of both the parsing stage using mika_ada_parser and the test input generation using mika_ada_generator within the extension. Once the comments have been entered and the dummy procedure implementation and calls have been added to the copied file the next step will be to run the parser on the file.

To do this the user must supply their path to both Mika's bin folder and gnat's bin folder located by default at:

- `C:\Users\[username]\AppData\Roaming\Midoan\Mika\bin`
- `C:\GNAT\2010\bin`

A settings page for the extension has been added where the user can update their particular paths.



mika-annotations-ada--js-

Gnat Path

C:\GNAT\2010\bin

Mika Path

C:\Users\derry\AppData\Roaming\Midoan\Mika\bin

Figure 28 - The extensions setting options for obtaining GNAT & Mika path if non defaults paths are used

The extension will perform a check to make sure both paths provided exist before continuing on.

```
var config =
vscode.workspace.getConfiguration("mika-annotations-ada--js-");

        var mp = config.mikaPath;

        var gp = config.gnatPath;

        if(fs.existsSync(mp) && fs.existsSync(gp)) {

            var terminal = vscode.window.createTerminal();

            terminal.show();

            terminal.sendText(`cd ${mikaFolder}`);


terminal.sendText(`${mp}${separator}mika_ada_parser.exe
-M"${mp}" -f"${gp}" -gnat05 -d ${nameMinusExt}`);

            terminal.sendText(`cd
${mikaFolder}${separator}${nameMinusExt}_mika`);
terminal.sendText(`${mp}${separator}mika_ada_generator.exe
-M"${mp}" -SMaximum -Tbranch -Cignored -d ${nameMinusExt}`);

        }
```

Figure 29 - Checks performed to make sure both paths exist before progressing

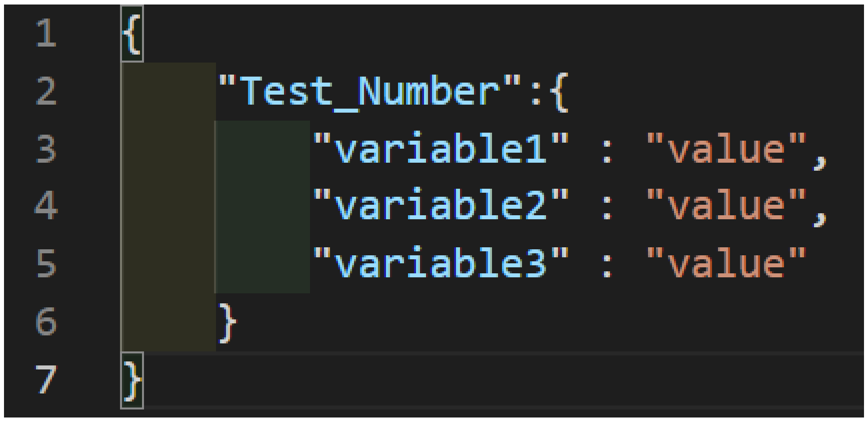
The first command to the terminal enters the newly created folder and once there calls on the mika_ada_parser located in Mika's bin folder, passing in the mikaPath with -M flag and gnatPath with -f flag, the gnat version to be used, here hard coded as -gnat05 and -d flag to enable debug mode along with the name of the program to be parsed. Javascripts string interpolation was used here to reuse the names of programs and paths which had previously been found during the creation of the new folder containing the files in question.

Once the parser has finished its process we next move into the folder that the parser created, called [programName]_mika. This contains the generated prolog file needed for the test input generator to run.

Then the call to the mika_ada_generator takes place, again requiring the path to Mika's bin folder with the -M flag, the subprogram in question with the -S flag, the type of coverage required with the -T flag and whether or not to ignore the elaboration with the -C flag and again using the -d flag to enable debug along with the name of the program.

The parsing of the program works well and creates the needed folder with the generated prolog code and other relevant files as discussed in the Mika User manual.

Moving on to how to present the outcome of the test results to the developer; the thought had been to have the generator return the results in JSON format such as:

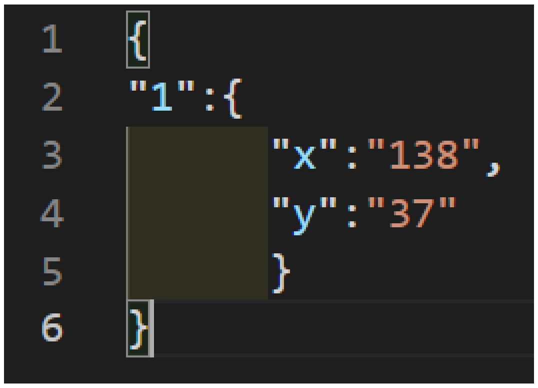


```
{
    "Test_Number":{
        "variable1" : "value",
        "variable2" : "value",
        "variable3" : "value"
    }
}
```

Figure 30 - Example format of the json file used to relay results from Mika to the extension

How to provide the developer details on the test inputs is still unclear, either in a file created back in their original working directory or with a pop up in Visual Studio Code itself. The final decision on how to display the test inputs back to the developer will be to display them in a tab next to the source code under test. This file can then be saved by the developer to a location of their choosing through Visual Studio Code.

Now that the parsing and generation are working from within the Visual studio code extension, work continues on with the idea of how to display the resulting output from Mika. As shown above a JSON format was chosen which would have the test number and the variable and values contained within it that caused the provided condition to be true.

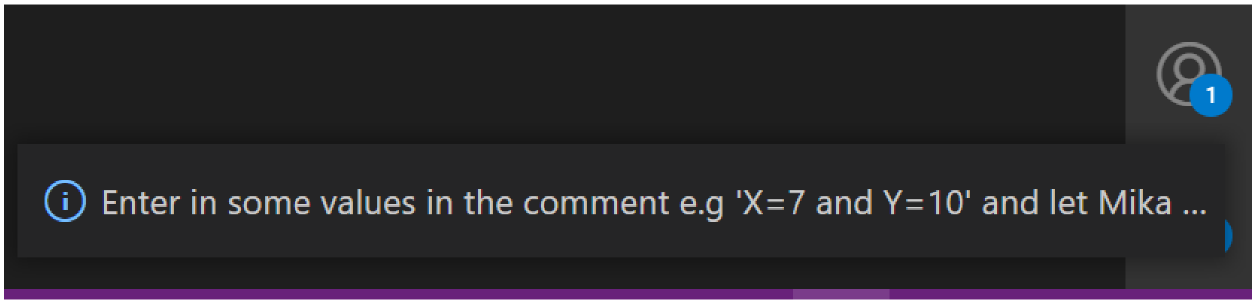


```
{
"1":{
        "x":"138",
        "y":"37"
        }
}
```

Figure 31 - Json file with example values

A mockup of the JSON file was created to allow testing until the real one could be generated.

With this in mind a decision had to be made as to how to display the tests back to the user. A pop-up seemed like the best fit, but the pop ups in Visual Studio Code are very small and would not be able to convey enough information to the user to be helpful. As seen below they are really just for important information and error messages.



Enter in some values in the comment e.g 'X=7 and Y=10' and let Mika ...

Figure 32 - A pop up explanation that was eventually removed in favour of the readme file

The next choice of display was to open another tab beside the annotated source code and to display the generated test inputs here, this way the developer can see both the original source code and cross reference it with the output from Mika.

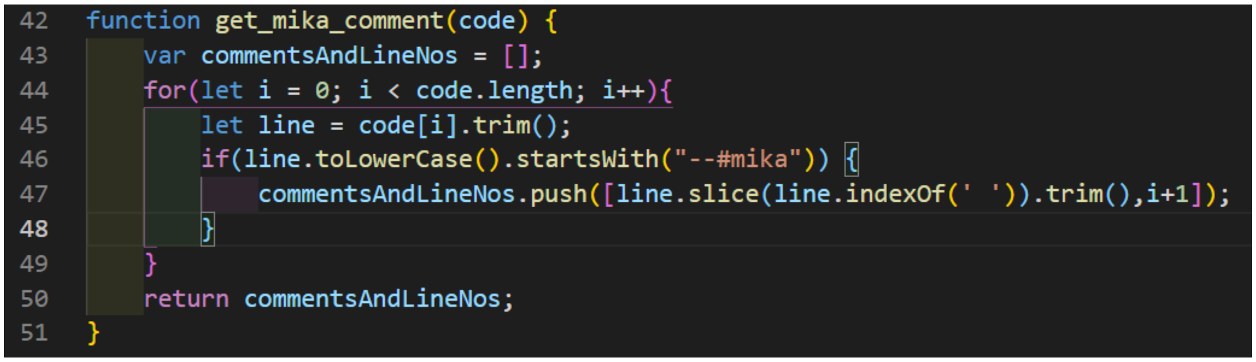
As of now this file is not saved and it can be saved by the developer in a location of their choosing if they so wish it.

To accomplish this the original source file has to be annotated with a special comment to get input from the developer, in this case "--#MIKA" is being used, this is then followed by a valid boolean statement such as X = 15.0 or X > Y. There is a command within the extension called "Mika Ada annotations", this will insert a boilerplate comment on the highlighted line, "--#MIKA 'enter the conditions you would like to be met here'". Once the developer has entered the required number of comments, the validity of the syntax in the provided comment is up to the developer, they can run the next command "Mika generate test inputs".

Mika generate test inputs is a multi process command. The first step is to gather the Mika and Gnat paths from the extensions settings which can be found under file -> preferences -> settings -> extensions -> mika-annotations-ada--js- . There are default paths provided but the developer will need to provide accurate paths to these folders for the extension to function properly. A check is conducted before running any command line instructions as to whether the provided paths exist.

The next step is making a copy of the current directory of the file open in VS code, this just copies all the files in the current directory to account for any dependencies to the file under test but does not copy in additional folders.

This folder is named "SecretMikaFolder" and all the manipulations of the file under test will happen on this copied version. The file open in VS code is scanned and the comments and lines.



```
function get_mika_comment(code) {
    var commentsAndLineNos = [];
    for(let i = 0; i < code.length; i++){
        let line = code[i].trim();
        if(line.toLowerCase().startsWith("--#mika")) {
            commentsAndLineNos.push([line.slice(line.indexOf(' ')).trim(),i+1]);
        }
    }
    return commentsAndLineNos;
}
```

Figure 33 - The code behind parsing the source code for the presence of --#Mika comments

This takes the boolean equation from the comment and the line number it was found on and returns this container to the main function.

4. Targeted Hardware/Software

The targeted platform is Windows 10 as Mika is untested on other operating systems.

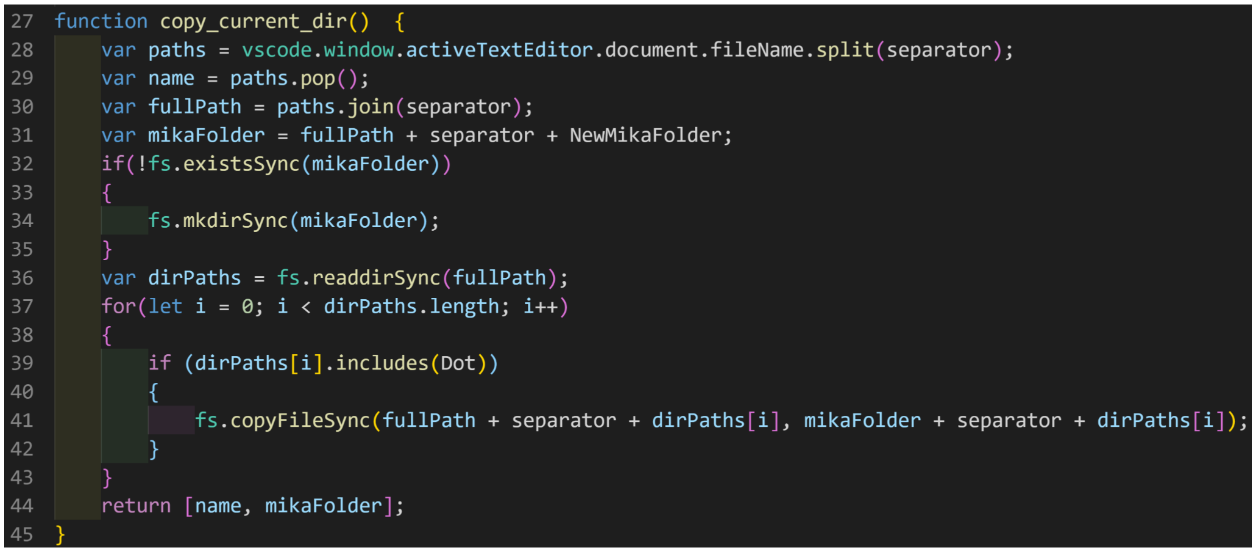
Dependencies of a GNAT Ada compiler and SICStus Prolog are also required for the correct compilation of Ada code and the operation of Mika.

To use the Visual Studio Code Extension, Visual Studio code must be installed and the paths to both Mika and GNAT added to the extensions settings page if other than default paths were selected for either installation.

5. Algorithms

A number of algorithms were implemented in the application of the Visual Studio Code extension for Mika. Below these will be shown screen shots of each with a brief explanation of what is occurring.

5.1 Copy current Directory



```
function copy_current_dir()  {
    var paths = vscode.window.activeTextEditor.document.fileName.split(separator);
    var name = paths.pop();
    var fullPath = paths.join(separator);
    var mikaFolder = fullPath + separator + NewMikaFolder;
    if(!fs.existsSync(mikaFolder))
    {
        fs.mkdirSync(mikaFolder);
    }
    var dirPaths = fs.readdirSync(fullPath);
    for(let i = 0; i < dirPaths.length; i++)
    {
        if (dirPaths[i].includes(Dot))
        {
            fs.copyFileSync(fullPath + separator + dirPaths[i], mikaFolder + separator + dirPaths[i]);
        }
    }
    return [name, mikaFolder];
}
```

Figure 34 - Copy current directory function

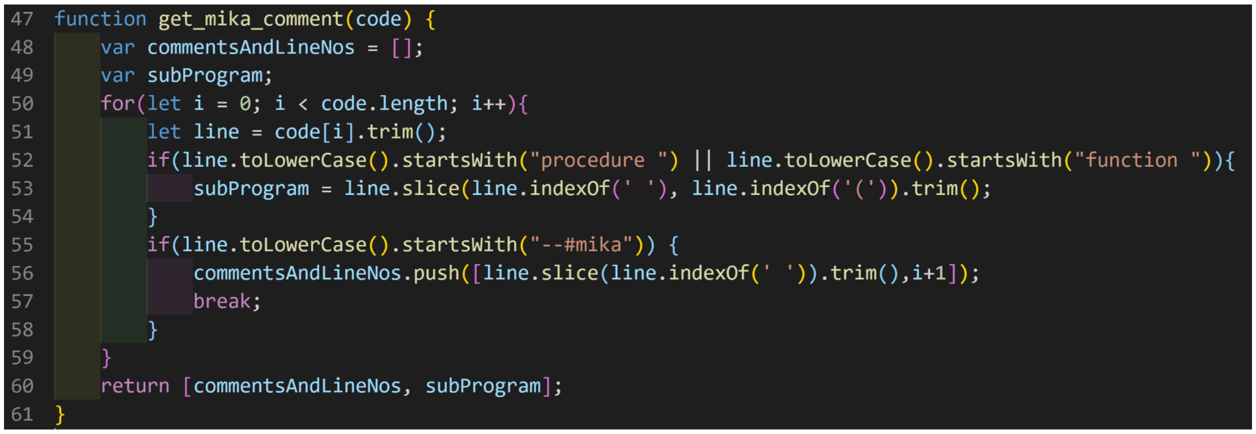
This function takes the full path of the current file open in Visual Studio Code (VS Code) and stores each section of the path in a list by splitting on the '\' separator.

The name of the current file is obtained by popping from the paths list. The rest are then rejoined again on the '\' again and a new separator and new Mika Folder name are also joined to the path.

If this new folder does not already exist it is created. All the non folder files in the current path are then copied into the new folder to maintain the integrity of the source code, any alterations to the code will happen on the copied files.

Once all operations are complete this function returns a tuple with the name of the file being operated on in the 'name' variable and the full path to the new Mika folder that was just created in the 'mikaFolder' variable.

5.2 Get Mika Comment



```
function get_mika_comment(code) {
    var commentsAndLineNos = [];
    var subProgram;
    for(let i = 0; i < code.length; i++){
        let line = code[i].trim();
        if(line.toLowerCase().startsWith("procedure ") || line.toLowerCase().startsWith("function ")){
            subProgram = line.slice(line.indexOf(' '), line.indexOf('(')).trim();
        }
        if(line.toLowerCase().startsWith("--#mika")) {
            commentsAndLineNos.push([line.slice(line.indexOf(' ')).trim(),i+1]);
            break;
        }
    }
    return [commentsAndLineNos, subProgram];
}
```

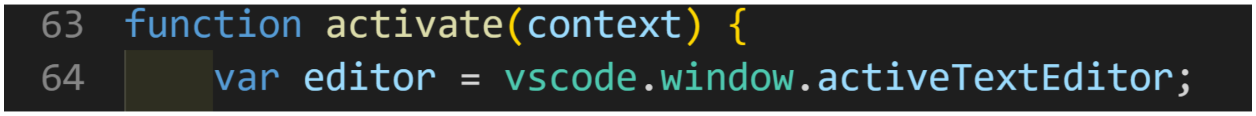
Figure 35 - Get Mika comment function

In the get_mika_comment function we are looking for the subprogram name the comment was inserted in and the boolean condition from the comment, the line number the comment was entered on is also recorded.

Each procedure or function name encountered is stored in the 'subProgram' variable and once the mika comment is encountered everything after the "--#mika " is stored as the comment and the index is stored as the line number.

The tuple of the commentsAndLineNos and subProgram are then returned.

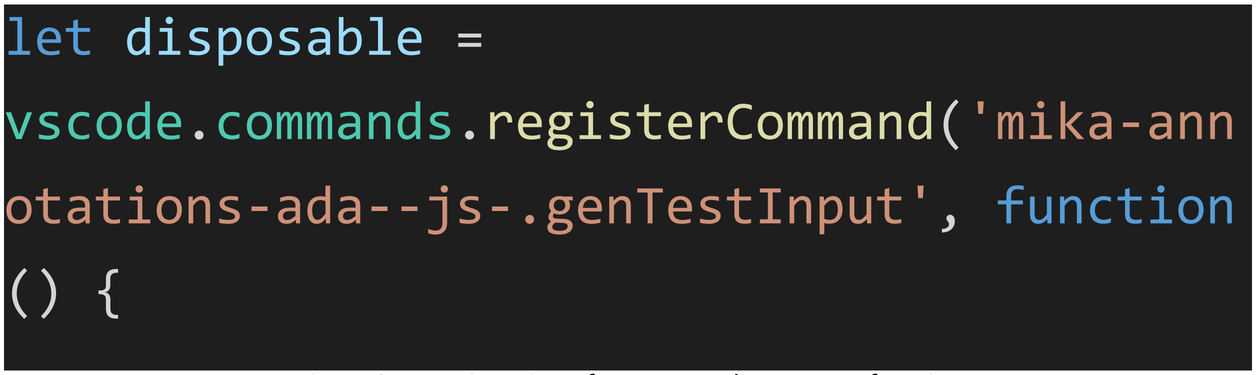
5.3 Activate



```
function activate(context) {
    var editor = vscode.window.activeTextEditor;
```

Figure 36 - Activate Function within the extension

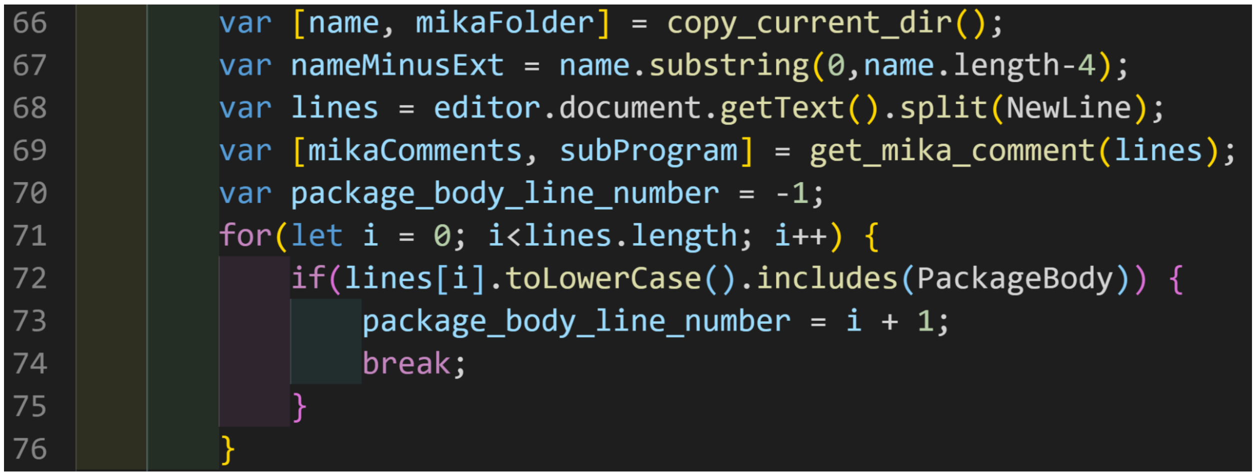
The activate function is the main function of the extension, inside of it two more functions are declared for the two commands available.



```
let disposable =
vscode.commands.registerCommand('mika-ann
otations-ada--js-.genTestInput', function
() {
```

Figure 37 - Registration of a command name to a function

The function above handles the code run when the Mika Generate test inputs are run.



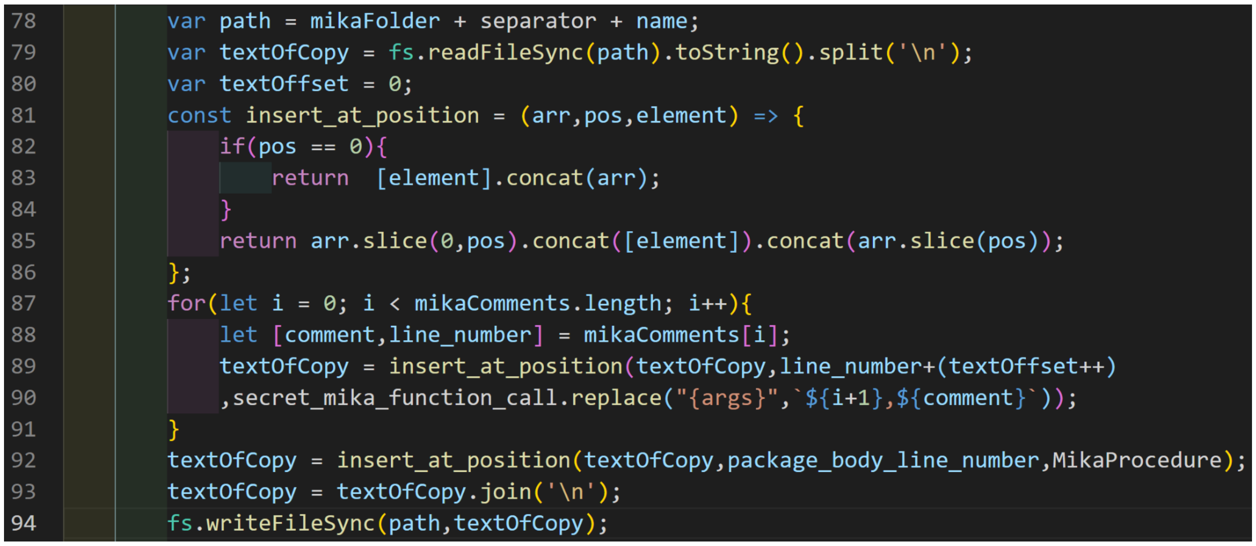
```
        var [name, mikaFolder] = copy_current_dir();
        var nameMinusExt = name.substring(0,name.length-4);
        var lines = editor.document.getText().split(NewLine);
        var [mikaComments, subProgram] = get_mika_comment(lines);
        var package_body_line_number = -1;
        for(let i = 0; i<lines.length; i++) {
            if(lines[i].toLowerCase().includes(PackageBody)) {
                package_body_line_number = i + 1;
                break;
            }
        }
```

Figure 38 - Start of the Generate test input command

The Generate test inputs command starts by calling the copy_current_dir function, which returns the name of the file being worked on and the path to the newly created Mika folder inside its containing folder. The nameMinusExt variable saves the name of the file after removing its extension e.g ".adb".

The code is then stored in an array by splitting the code on each '\n' (new line) found within its text. And this is parsed over by the get_mika_comment function which returns the subprogram the comment was entered into and the boolean condition provided.

The code is also iterated over looking for the line number at which the package body declaration is found in order to appropriately insert a new procedure later on.



```
var path = mikaFolder + separator + name;
var textOfCopy = fs.readFileSync(path).toString().split('\n');
var textOffset = 0;
const insert_at_position = (arr,pos,element) => {
    if(pos == 0){
        return  [element].concat(arr);
    }
    return arr.slice(0,pos).concat([element]).concat(arr.slice(pos));
};
for(let i = 0; i < mikaComments.length; i++){
    let [comment,line_number] = mikaComments[i];
    textOfCopy = insert_at_position(textOfCopy,line_number+(textOffset++)
    ,secret_mika_function_call.replace("{args}",`${i+1},${comment}`));
}
textOfCopy = insert_at_position(textOfCopy,package_body_line_number,MikaProcedure);
textOfCopy = textOfCopy.join('\n');
fs.writeFileSync(path,textOfCopy);
```

Figure 39 - Code within generate test input that handles the insertion of secret Mika procedure and calls

The path to the copied file is stored in the variable 'path', the text of the copied file is then saved in the variable textOfCopy using the same method applied earlier.

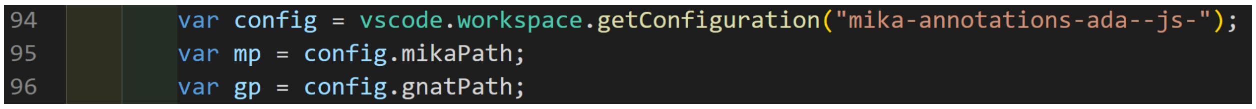
5.3.1 Insert_at_position

A function insert_at_position is declared to handle inserting extra lines into the array of code lines saved earlier. This function takes an array (arr), the position to insert at (pos) and the element to insert (element).

If the position is 0 we return the array with the new element concatenated to the start of it, otherwise we return an array sliced at the position provided with the element concatenated between the two slices.

The array of comments is iterated over and a new procedure call is inserted into the code at the line number the comment was found at +1. (Currently the extension only supports one comment per file but this list of comments is a future proofing method designed to handle multiple comments at a later stage also).

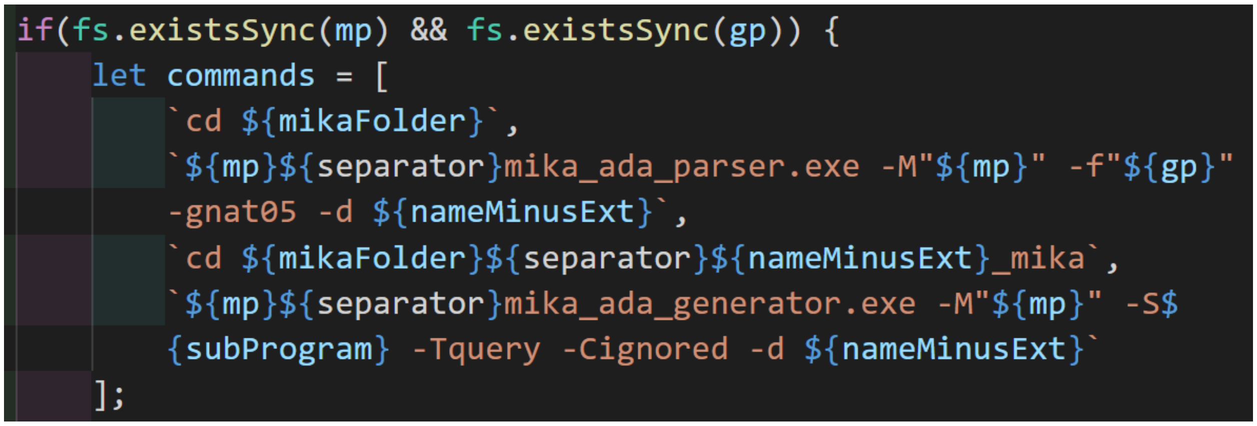
The new procedure is then inserted after all the comments have been handled in order to not skew the line numbers the comments were found at. And the array is joined back up on the new line character and resaved as the copy of the file under test.



```
        var config = vscode.workspace.getConfiguration("mika-annotations-ada--js-");
        var mp = config.mikaPath;
        var gp = config.gnatPath;
```

Figure 40 - Acquiring Paths from extensions settings

The paths to both mika and GNAT are obtained from the extensions settings, the default paths for both Mika and GNAT are saved here initially, a user who has them saved in a path other than default will need to update these settings manually.

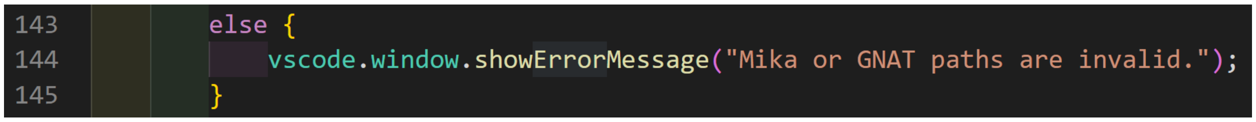


```
if(fs.existsSync(mp) && fs.existsSync(gp)) {
    let commands = [
        `cd ${mikaFolder}`,
        `${mp}${separator}mika_ada_parser.exe -M"${mp}" -f"${gp}"
        -gnat05 -d ${nameMinusExt}`,
        `cd ${mikaFolder}${separator}${nameMinusExt}_mika`,
        `${mp}${separator}mika_ada_generator.exe -M"${mp}" -S$
        {subProgram} -Tquery -Cignored -d ${nameMinusExt}`
    ];
```

Figure 41 - Commands used to call Mika stored in an array

If both paths exist an array of commands is defined, these commands are the CLI commands Mika needs to run the dynamic code query on the file and subprogram in question.

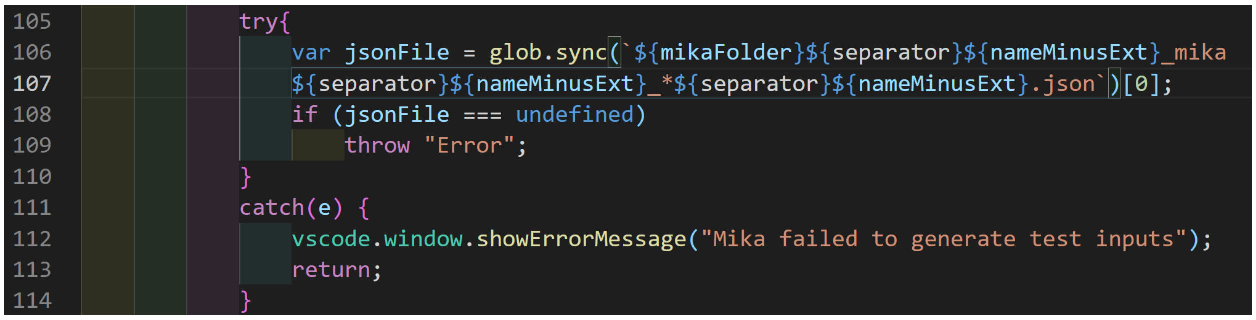
The node package child_process (cp) is then used to run these commands synchronously in a shell where each command is run one after another using the join on ‘ & ‘. this will allow this process to finish running before the program proceeds as the output of these commands is need as the program proceeds.



```
        else {
            vscode.window.showErrorMessage("Mika or GNAT paths are invalid.");
        }
```

Figure 42 - Error message displayed to user if Either path does not exists

If either of the paths are invalid an error message is displayed to the user stating as such.

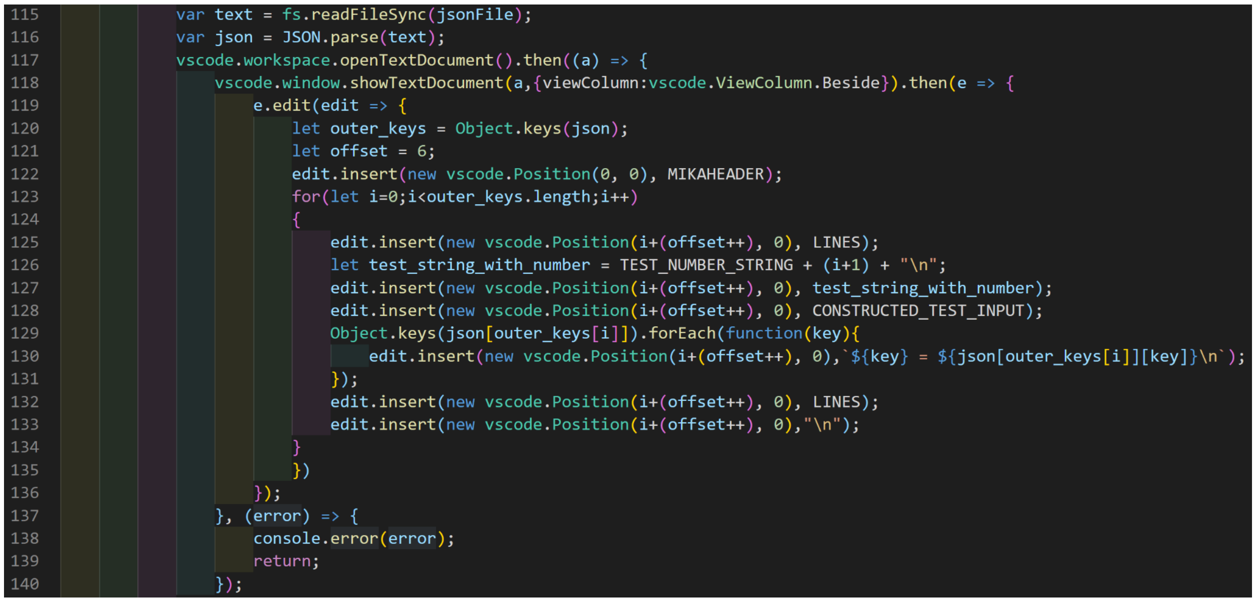


```
try{
    var jsonFile = glob.sync(`${mikaFolder}${separator}${nameMinusExt}_mika
    ${separator}${nameMinusExt}_*${separator}${nameMinusExt}.json`)[0];
    if (jsonFile === undefined)
        throw "Error";
}
catch(e) {
    vscode.window.showErrorMessage("Mika failed to generate test inputs");
    return;
}
```

Figure 43 - Using Glob to search for newly created folder with date/time stamp in its name

Next, once the shell commands have finished running the glob package is used to look for the produced folder which has a date/time appended to the name of the file, glob is a pattern matching package for files, here we are looking for the JSON file contained within this generated folder.

If the json file is not present an error message is returned stating that Mika failed to generate test inputs.

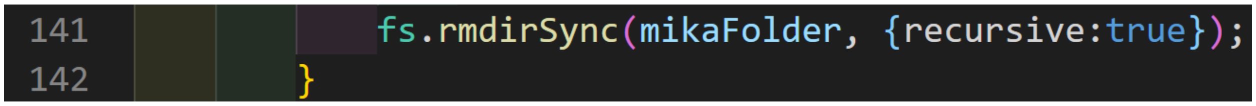


```
var text = fs.readFileSync(jsonFile);
var json = JSON.parse(text);
vscode.workspace.openTextDocument().then((a) => {
    vscode.window.showTextDocument(a,{viewColumn:vscode.ViewColumn.Beside}).then(e => {
        e.edit(edit => {
            let outer_keys = Object.keys(json);
            let offset = 6;
            edit.insert(new vscode.Position(0, 0), MIKAHEADER);
            for(let i=0;i<outer_keys.length;i++)
            {
                edit.insert(new vscode.Position(i+(offset++), 0), LINES);
                let test_string_with_number = TEST_NUMBER_STRING + (i+1) + "\n";
                edit.insert(new vscode.Position(i+(offset++), 0), test_string_with_number);
                edit.insert(new vscode.Position(i+(offset++), 0), CONSTRUCTED_TEST_INPUT);
                Object.keys(json[outer_keys[i]]).forEach(function(key){
                    edit.insert(new vscode.Position(i+(offset++), 0),`${key} = ${json[outer_keys[i]][key]}\n`);
                });
                edit.insert(new vscode.Position(i+(offset++), 0), LINES);
                edit.insert(new vscode.Position(i+(offset++), 0),"\n");
            }
            })
        });
    }, (error) => {
        console.error(error);
        return;
    });
```

Figure 44 - Generation of the new tab displaying results from Mika to the User

Once the Json file has been located and parsed the code above handles the creation of the new tab to be displayed with the results. A variable offset is assigned a value of 6 to account for the header to be applied to the file, the rest handles the insertion of the test inputs found within the JSON file.

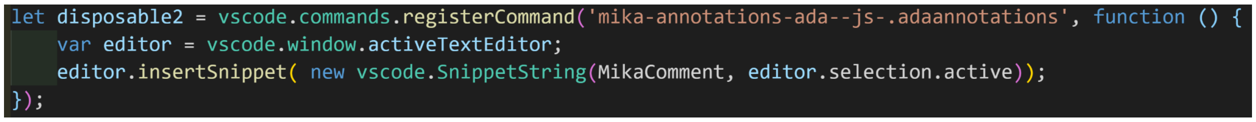
Again this was designed with multiple comments in mind, but will handle the single comment currently supported as well.



```
            fs.rmdirSync(mikaFolder, {recursive:true});
        }
```

Figure 45 - Removal of the directory containing copied files used for input generation

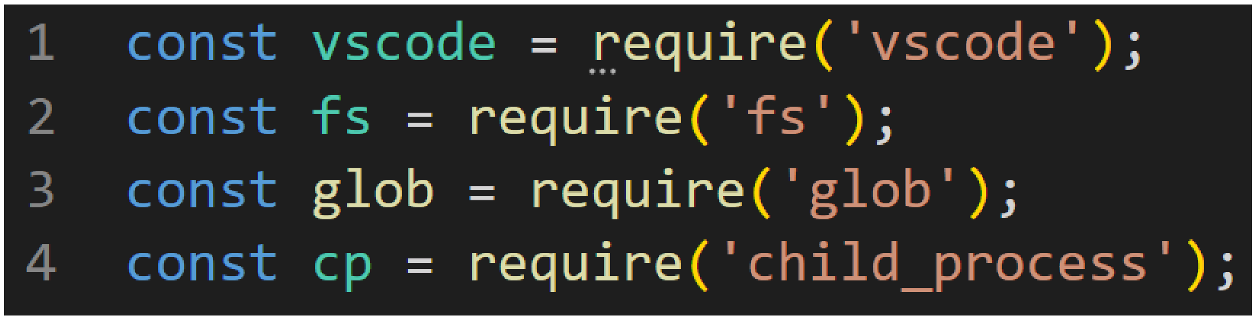
Finally the newly created folder containing the copies of the files and the generated files is removed, leaving the developer to save the displayed test inputs in a location of their choosing using VS code.



```
let disposable2 = vscode.commands.registerCommand('mika-annotations-ada--js-.adaannotations', function () {
    var editor = vscode.window.activeTextEditor;
    editor.insertSnippet( new vscode.SnippetString(MikaComment, editor.selection.active));
});
```

Figure 46 - Simple command used to insert boilerplate Mika comment at highlighted line in the text editor

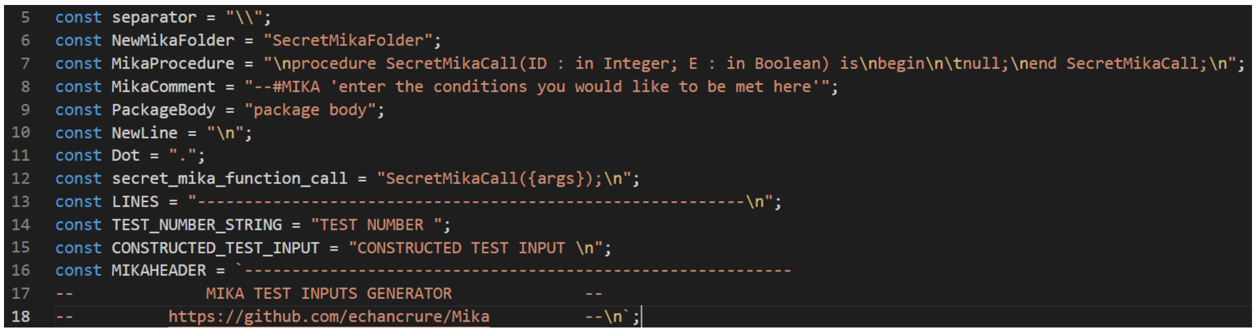
The function above handles the code when the Mika Ada annotations command is run. As can be seen the current position of the cursor is taken from the active text editor and then the boilerplate comment is inserted at that line.



```
const vscode = require('vscode');
const fs = require('fs');
const glob = require('glob');
const cp = require('child_process');
```

Figure 47 - Required Node.js packages for the extension

These are the node.js packages used within the extension, the vscode package allows direct access to the VS code text editor, fs (filesystem) allows for the creation and deletion of files and directories, glob is a file pattern matching package allowing for the selection of a file we do not know the exact path to due to generated date/time names on a folder in the path to that file and the child_process package creates a new child process to the extension process allowing for the synchronous running of the terminal commands within that child process.



```
const separator = "\\";
const NewMikaFolder = "SecretMikaFolder";
const MikaProcedure = "\nprocedure SecretMikaCall(ID : in Integer; E : in Boolean) is\nbegin\n\tnull;\nend SecretMikaCall;\n";
const MikaComment = "--#MIKA 'enter the conditions you would like to be met here'";
const PackageBody = "package body";
const NewLine = "\n";
const Dot = ".";
const secret_mika_function_call = "SecretMikaCall({args});\n";
const LINES = "----------------------------------------------------------\n";
const TEST_NUMBER_STRING = "TEST NUMBER ";
const CONSTRUCTED_TEST_INPUT = "CONSTRUCTED TEST INPUT \n";
const MIKAHEADER = `---------------------------------------------------------
--              MIKA TEST INPUTS GENERATOR              --
--          https://github.com/echancrure/Mika          --\n`;
```

Figure 48 - Constant values for Strings used within the extension

A number of constant strings were also declared in order to keep the algorithm's code tidy and not cause any unnecessary slow down during operation.

References

[1] Core, A., 2021. About Ada. [online] AdaCore. Available at: <https://www.adacore.com/about-ada> [Accessed 11 April 2021].