dOvs - Semantic Analysis

Group 9
Miran Hasanagić - 20084902
Jakob Graugaard Laursen - 20093220
Steven Astrup Sørensen - 201206081

October 7, 2015

1 Introduction

This report describes how the semantic analysis was developed. The semantic analysis is the third component of the compiler. This component takes the Abstract Syntax Tree (AST) from the parser and makes a semantics analysis.

First it is described how the missing cases where implemented in the provided file `semant.sml` and which other files had to be extended in order to developed the semantic analysis component. Afterwards, the problems encountered and experience gain are described. Then 5 tiger program are presented, which test different cases of the semantics analysis. Finally, the work carried out and the experience gained are summarised.

2 The semantic analysis

The main working of the semantics analysis is that it takes the AST provided by the parser and applies the function `transExp`. This function is able to transvers a AST, and created a Type AST (TAST), based on the file `tabsyn.sml`. This new TAST, which is the output of the semantic analysis module, is a AST which has be decorated with type information. This was a high level description of how the semantic module is working. Below more low-level parts are highlighted.

The main development in order to create the semantics analysis done, was made inside the file `semant.sml`. Basically the approach from the book together with the adjustments from dOvs was followed during the development. For example the nested function `trexp` inside the function `transExp`, is used on all expressions which do not change the environment. Then `transExp` is only used when an expression has to be evaluated with an new environment. As an example this is the case when the body of let expression has to be evaluated with possibly new declarations. Also the `extra` is used in order to send additional needed information with the relevant functions.

Also a key part of this semantic analysis is the handling of the declarations, especially the recursive declarations of types and functions. In this case first the non-recursive version (part a) was developed. Afterwards, it was extended to handle recursive declarations. The recursive declarations are handled by going through them two times. For types, first all the headers of each type declaration are stored. In the next evaluation actual type declaration is identified. TODO: Explain the cycle identifier. The recursive function where handled in a similar way by analysing through two runs, by first going through the headers, and afterwards the body.

3 The support modules

The support module `ENV` was extended with the information about the standard library functions as described in Appendix A in the course book. TODO: Make examples.. This `ENV` module also was used as part of creating the environment for typs and values during the semantical analysis.

4 Problems encountered & experience gained

I key experience is how to create the semantic analysis in order to detect which parts are not working correctly during development. For this reason Test Driven Development was used, like in our work in the warm up assignment. During the development small unit test where created in order to test single aspects of the module isolated. This helped in order to identify the source of the error.

Additionally, it led to discussions about how to handle type error in order not to let an error propagate. The main principle in this approach is to ensure that the semantic analysis can continue even though an error was encountered. Such things can be handled in different ways, and here we had to make a compromise and decision.

5 5 tiger programs

These five tiger programs presented here are created simple in order to focus on different aspects of the implemented semantic analysis, which have not be covered by the provided testcases. These tests contain different syntax errors as required by the project description. In order to ensure that this semantic analysis works it is run on the provided testcases by the course.

5.1 break_exp_tests.tig

```
/* Check the three possible states for break.
        in for loop, in while loop, outside loop
        EXPECTION: an error at last break statement
 */
(for i := 1 to 3 do break; while 2 do break;break)
```

5.2 cycle_in_typedecl.tig

```
/* test recursive typedefintion */
let
        type A = B
        type B = D
        type D = A
        type t1 = t2
        type t2 = t3
        type t4 = t2
        type t3 = int
        type C = int
        type tree = {value: int, children:treelist}
        type treelist = {hd:tree, tl:treelist}
in
end
```

5.3 unassignable.tig

```
(for i := 1 to 3 do i:=2)
```

5.4 recursive_type_usage.tig

```
let
  type tree = {value:int, children:treelist}
  type treelist = {hd: tree, tl:treelist}
  var c := tree{value = 0, children = nil}
  var children := treelist{hd = c, tl = nil}
  var tree := tree{value = 1, children = children}
in tree
end
```

5.5 Test1

```
Just a test..
```

6 Conclusion

This report presented the development of the semantics analysis, which creates a typed AST. Additionally the gained experience was described. Finally, additional tests provided confidence that this semantic analysis is working correctly and also reports correct error messages.

This report showed the implementation of the parser. Furthermore some intersting problem encountered where described.