Phase 2: Semantic Analysis

Compiler Construction
Radboud University

In this second assignment you will implement a semantic analysis for SPL. This implementation is based on the abstract syntax tree produced by the parser from the first assignment. The semantic analysis should at least contain binding time analysis and type checking. You can decide whether you want to implement type inference, but you should at least try to do it. Type checking is easier, but type inference will be a plus point for your final grade.

1 Binding time analysis

Here you should check whether all applied occurrences of identifiers are associated with a proper definition. There is no formal definition of SPL, you have to make some decisions. The given examples and grammar provide hints for the decisions to be made.

It should be possible to use functions and global variables before they are defined.

There are three kinds of variables in SPL:

1. Global variables.
2. Function arguments, which can be used like local variables in the function body.
3. Local variables, which can be defined at the beginning of a function body.

In most programming languages local variables hide function arguments and global variables with the same name. Function arguments hide global variables with the same name. It is sensible to follow these rules in SPL.

Since SPL only has first order functions, the namespaces for functions and variables are disjoint. It should not be a problem to use functions with the same name as variables.

Enforcement of scoping rules is usually not a separate phase in the semantic analysis, but a consequence of how environments are handled during type checking. Think about how you want to deal with name clashes and make sure that your implementation behaves accordingly. Do you want to produce warnings when a name hides another? Decide and document it! Write tests!

2 Type checking

If you implement type checking, type annotations for functions cannot be optional, and the **var** keyword for variable declarations cannot be used. You have to change the grammar for that.

Design and implement typing rules for expressions, statements and functions in SPL.

In an imperative language like SPL you need to do more than just check if every expression is well typed. For instance, you need to check that a **Void** function never returns a value, and that a function that should return a value indeed returns a value of the proper type in every code path. It does help when you know the proper binding of variables and functions.

3 Polymorphism

Functions can be polymorphic, for example

```
swapCopy( pair ) :: (a, b) → (b, a)
{
    return (pair.snd, pair.fst);
}
```

Polymorphic functions work for arguments of any type.

4 Overloading

Some operations are overloaded, for example the equality operator (==) :: a a → **Bool** can compare two values of the same type, and **print** :: a → **Void** can show values of any type. Most likely you need different implementations of these functions for different types. Your compiler needs to find a way to select the appropriate implementation based on the type of the argument.

Are there situations in SPL where overloading cannot be solved at compile time, for instance can an overloaded operation be used on polymorphic arguments?

Since type information is needed during code generation, it is not sufficient to just check that programs are well typed. You also need to decorate the syntax tree with type information so that code generation can make use of it. Of course you don't have code generation yet, so just let your pretty printer print the decorated syntax tree.

Producing fancy error messages is not required, just indicate that there is a problem and stop compilation.

5 Type inference

The grammar of SPL allows leaving out type annotations for functions. Furthermore, when declaring a local or global variable, the **var** keyword can be used instead of a type.

```
swapCopy ( pair ) {
  var x = pair.fst;
  var y = pair.snd;
    return (y, x);
}
```

In such cases, your semantic analysis has to perform type inference. If you implement type inference and a function or variable has a type annotation, your compiler always infers the type and just compares it with the given type.

Deliverables

You have to give a presentation about your type checker, like you did for the parser. The date has been announced in the lectures and on Blackboard. By now we are all SPL experts, so focus on what is specific for your compiler. Describe what you implemented and how you did it. Which problems did you encounter and how did you solve them? Show us some non-trivial example programs and how your compiler deals with them. This demonstration doesn't need to be live, you can put input and output on the slides.

Don't forget that your final report must have a section about the semantic analysis. It is a good idea to make notes now when your memory is still fresh. The final report basically contains what you said in your presentation, but in more detail. Describe what you analyse and how you implemented it. How do your error messages look like? Mention problems you encountered and how you solved them. How does polymorphism, type inference and overloading work together? Include at least five working example programs and five that produce error messages, and describe why or why not your type checker accepts them. Are there programs that you would like your type checker to accept, but for some reason doesn't?