# Phase 2: Semantic Analysis

Compiler Construction
Radboud University

February 26, 2019

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);
}
```

## 6 Decisions

To summarize, you have to implement one of the following levels of semantic analysis. The levels are in increasing order of difficulty. We are perfectly happy if you only implement the first level.

1. Monomorphic type checking
2. Polymorphic type checking
3. Monomorphic type inference
4. Polymorphic type inference

None of those systems initially supports mutual recursion. If you want mutual recursion, you can add it to any system, no matter which one you choose.

### 6.1 Monomorphic Type Checking

Type checking means that the programmer always has to specify type signatures for all functions and variables. Monomorphic means that there are no type variables. For your compiler, this means that you change the grammar of variable and function declarations. Remove the **var** keyword and identifiers from types. Type signatures for functions are mandatory. The changed rules in the grammar are as follows.

```
VarDecl    = Type id   '=' Exp ';'
FunDecl    = id '(' [ FArgs ] ')' '::' FunType '{' VarDecl* Stmt+ '}'
Type       = BasicType
           | '(' Type ',' Type ')'
           | '[' Type ']'
```

Type checking is depth-first traversal of the abstract syntax tree. The types of leaves are always known, because leaves are either constants like numbers or booleans, or variables, in which case the type can be found in the environment. The type of inner nodes can be checked by first recursing into the children and using the result to determine the type of the inner node.

In this language it is not possible to express the polymorphic identity function.

```
// In monomorphic type systems, every type needs its own identity function
idInt ( x ) :: Int → Int {
  return x;
}

idBool ( x ) :: Bool → Bool {
  return x;
}
```

Hint: You might want to introduce a special internal type for the empty list that is compatible with any other list. This helps when type checking expressions like the following.

```
[([Int], [(Int, Bool)])] x = ([], (1, True) : (2, False) : []) : [];
```

## 6.2 Polymorphic Type Checking

Polymorphic type checking still requires programmers to always specify type signatures, but type signatures can contain type variables. The changed rules in the grammar are as follows.

```
VarDecl    = Type id   '=' Exp ';'
FunDecl    = id '(' [ FArgs ] ')' '::' FunType '{' VarDecl* Stmt+ '}'
Type       = BasicType
           | '(' Type ',' Type ')'
           | '[' Type ']'
           | id
```

In this language it is possible to write polymorphic functions, but the programmer always has to specify type signatures. Type checking is still depth-first tree traversal, and types are just compared for equality. Type variables are nothing special, the type checker treats them like base types. Different type variables are different types. You must implement generalization and instantiation, but only for global functions. The value restriction still applies, which means you never generalize local variables.

```
// Internally, the type gets generalized to ∀a.a → a
id(x) :: a → a {
  return x;
}

// You should now be able to write the polymorphic list reversal
reverse(list) :: [a] → [a] {
  // ...
}

main()
{
  // The type of id must be instantiated differently for each occurrence
  print(id(5));
  print(id(True));
}

// This function should give a type error
```

```
const(x, y) :: a b → a {
  return y;
}
```

## 6.3 Monomorphic Type Inference

The programmer can leave out type signatures. This means your type inference has to work with fresh type variables, unification and substitutions, but type variables are never generalized. The changed rules in the grammar are as follows.

```
VarDecl   = Type id  '=' Exp ';'
FunDecl   = id '(' [ FArgs ] ')' [ '::' FunType ] '{' VarDecl* Stmt+ '}'
Type      = BasicType
          | '(' Type ',' Type ')'
          | '[' Type ']'
```

In this language, the type checker detects for example that the factorial function is of type **Int** → **Int**.

```
// Look ma, no type signature!
facR(n) {
    if ( n < 2 ) {
        return 1;
    } else {
        return n * facR(n-1);
    }
}
```

The following program however leads to a type error, because the identity function can not be polymorphic.

```
id(x) { return x; }

main() {
  print(id(10));
  print(id(True));
}
```

## 6.4 Polymorphic Type Inference

Functions can be polymorphic, and the types of local variables can be inferred. You need all the machinery you have seen in the lecture with substitutions, unification, generalization and instantiation. The following program should typecheck and work as expected.

```
id(x) { return x; }

main() {
  var x = 10;
  var y = x;
  print(id(y));
  print(id(True));
}
```

# 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 Brightspace. 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?