Compiler Construction Final Report

Flip van Spaendonck & Lars Kuijpers
s4343123 & s4356314

June 13, 2018

Contents

1 Introduction

1.1 Programming language

1.2 Error Handling

1.3 Final Grammar

2 Lexer

3 Parser

3.1 Structure

Now that we have the list of tokens that is the lexed code, we can start running our parser on it. Out parser runs on a bottom-up architecture, to do this we first give the grammar/syntax of the SPL language to our compiler which turns it into a graph. This graph is used by the compiler to check which tokens are allowed at what point in the code.

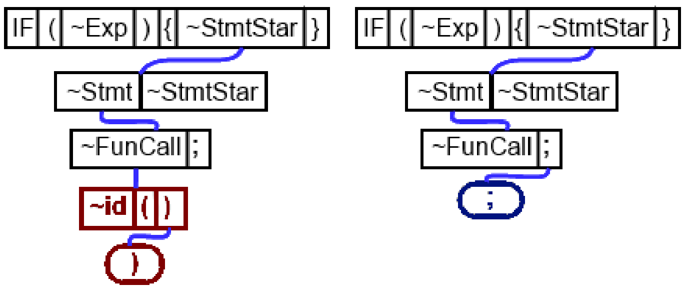

Figure 1: Example of two tokentraces

If at some point there are no possibilities left, but there are still tokens left to be parsed, the compiler will give an error to the user, telling it that their code is uncompilable. However when a token is allowed, it is mapped to a tokentrace. *(See figure 1)* A token trace consists of a list of nested expressions, the token that we just parsed and a list of tokentraces that cover the tokens left of this token in the code. So in the right trace in figure 1: a funcall inside statement composition inside an if statement, the ";" and the tokentraces that went before this one (which includes the tokentrace on the left).
Through this we can get a time-complexity of $\mathcal{O}(n \cdot k)$ with n being the number of tokens in the code, and k being the maximum amount of possible tokens that could follow a certain token. While n can of-course vary per piece of code, k is heavily influenced by how the syntax is written down.
Once all tokens of the code have been correctly parsed and the last tokentrace itself doesn't expect a next token, we will have found a correct way of parsing the code. If multiple ways have been found of parsing the same code, this could for example happen when parsing expression only containing a variable, the

first possibility is taken, it is up to the structuring of the grammar itself to make sure that this can't lead to two possible parses with differing semantics.
Now that we have a list of tokentraces, the next step is putting them together into a syntax tree.*(See figure 2)* No errors should be able to occur while the syntax tree is being constructed.
Once we have constructed the syntax tree, we will convert it into a partially abstract syntax tree. This is done by creating a new ast whilst moving through the tree and checking for each node/expression whether it has an ast equivalent, if it does the equivalent ast node is constructed and added to the new tree, if not a copy of the node is added to the new tree.

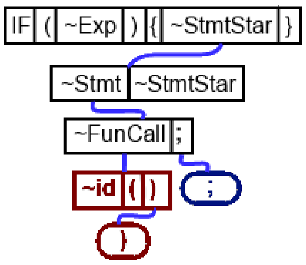

Figure 2: Part of a syntax-tree consisting of the two tokentraces described in 1

3.2 Expression Optimization

To reduce time-complexity we parse actual expression separately. Before the above described parsing occurs, the code is first processed, detecting the locations of where an expression should be e.g., when parsing "print 4+5*3;" it will detect the print keyword and then take all tokens between the print keyword and the next ";" i.e. "4+5*3". The extracted tokens are then turned into custom expression tokens, which immediately parse them using the EXP grammar/syntax. The then parsed expressions are then emmediately checked for any redundant steps, i.e. "4+5*3" will always return 19, thus it is reduced to a simple constant, instead of the more complicated extraction of before.
When the actual code is then eventually parsed the syntax just expects an expression token instead of an actual expression.

4 Semantic Analysis

The next step in compiling our code will be semantic analysis. As previously described we've transformed our syntax tree into on with ast nodes.
Currently the only semantic analysis that is done, is checking if everything is well-defined and well-typed, and checking whether every non-void function will eventually hit a return statement.

4.1 Type-checking and checking for well-definedness

Checking if everything is well-typed and well-defined is done in the same step. We do this by traversing the tree and checking the following for each node:

- Whether the nodes declares anything new e.g., a Variable declaration.
- Whether the node is well-typed.
- Whether any undefined id is used.

The order in which this is done and how this is done, is entirely dependent and taken care of per node e.g., for a variable declaration it's own name is not allowed to be used in the declaration itself, while in a function declarations body it is allowed to use the function itself such that recursive functions can occur. The ID's that are defined and their types are stored inside of an IDDeclarationblock, this block is then passed through to the next node. If an id is not declared a declaration exception is thrown. If something is incorrectly typed, for example an integer is assigned to a variable of type boolean, or an expression of type tuple is used in an if-statement, a type exception is thrown. If an exception is thrown the compiler stops compiling and gives the error to the user.

4.2 Checking for termination

Checking whether a non-void function will eventually hit a return statement, is actually already done at the moment its ast node is generated. During generation the function's body is traversed to check, given that it terminates, will eventually hit a return statement. Checking whether the return statement is of the correct type is done later, as described above.

5 Code Generation

5.1 Basics

5.2 Codestack

5.3 Variables

5.4 Data structures

5.5 Function calls

6 Extension

6.1 Idea

6.2 Implementation

A Examples