# Creating a functional programming language based on System F

William Bodin Martin Fredin Samuel Hammersberg
Valter Miari Victor Olin Sebastian Selander


CHALMERS
UNIVERSITY OF TECHNOLOGY


GÖTEBORGS
UNIVERSITET

# Glossary

- System F - A typed lambda calculus that provides a theoretical basis for mathematical programming languages like Haskell and ML.
- Haskell - A general-purpose functional programming language popular in academia.
- BNFC - Backus-Naur Form Converter, a tool to generate lexers and parsers in a specified host language. These tools generated will help form the compiler for the language in question. [1]
- Lexer - The first part of the compilation phase which reads a given text and splits it into tokens. This phase creates the tokens necessary for the parser to continue the compilation.
- Parser - The second phase of compilation, receives tokens from the lexer and parses them into a syntax tree.
- AST - Abstract syntax tree, a tree data structure to represent the flow of a program.
- Annotated AST - An extension of an AST with type annotations on operations and data types.
- LLVM - A compiler suite with reusable tools to create compilers for modern languages. [2]
- LLVM IR - LLVM intermediate representation, a platform-independent high-level assembly language.
- Alex - A tool for generating lexical analyzers in Haskell.
- Happy - A parser generator for Haskell.

# 1 Background

In general, there have always been two distinct branches of programming, imperative and functional programming. Functional languages have several differences to imperative languages, they come with greater expressive power, a more declarative style, and a close resemblance to mathematics. The influence of functional programming can also be noticed in recent trends regarding non-functional languages, some, which have borrowed functionality from functional languages, like C# implementing Linq to support more functional styles of programming. Rust is a more recent programming language that has taken a lot of inspiration from functional languages such as Haskell and OCaml [3].

In functional languages, powerful type systems are formed on the basis of mathematics and typed lambda calculus such as System F [4]. This provides a way to express the behavior of programs and also catch developer-induced bugs already at compile time.

When designing a language, one has to face a critical decision regarding the execution of programs written in the language in question, whether the language should be interpreted or compiled. Interpreted languages have the advantage of inheriting the architectural support of the host language. Comparatively, the host language of the compiler, as well as the architecture of the generated code has to be supported.
While writing compilers can be tedious for the developer, it comes with the advantage of the compiled program being faster. When interpreters are executing a program, the interpreter and its runtime have to handle all communication with the hardware through the operating system. A compiled program can directly be executed by the operating system such that no middle layer exists between the hardware and the program.

LLVM [2] is a compiler infrastructure that allows one to compile a program to any of its supported target architectures. This compiler infrastructure will be used to avoid architecture-specific hardware and software details, such as reading registers and manipulating the stack. Taking this load off of the developer such that one can focus more on implementing the language rather than debugging machine code on various systems.

Clearly, programming language design is a complex task. It is important that different methods are tried and documented, allowing developers in the future to take advantage of the processes that worked well, and also not repeat the mistakes of previous language designers.

# 2 Aim

The two primary goals of the project are to explore the process of creating a statically typed functional programming based on System F that is compiled to LLVM [2], and explore some advanced concepts related to programming language theory. As such, two important aspect of the project is documentation as well as verifying the correctness of the implementation.
The compiler should be based on good and robust methods, this means that the methods used should, if possible, be based on previous successful attempts. One such example is basing our type rules solely on those of System F.

# 3 Scope

Implementing a useful general-purpose language with all the functionality users have become accustomed to is complex and time-consuming. Therefore, scoping, in the sense of; knowing what to prioritize and trying to define what is achievable within a certain time frame, is an integral part of the process.

The language is not supposed to be useful, but rather it is an academic exercise. One example of uselessness is the lack of effects, i.e. interactions with the outside world. The only planned effect is to print the computed value.

Another feature that is out of scope for this project is an extensive standard library. Many modern programming languages have a big collection of functions for performing common tasks. One example of this is the programming language Go, which is a programming language initially developed as a systems language with a focus on multi-processing [5]. Still, the first release of Go included a package for creating a web server. The language of this project will not focus on a standard library because there seem to be fewer interesting design decisions, in comparison to other parts of the compiler.

Other areas that will not be implemented:

- External library management
- Linking and modules
- Code optimization
- Concurrency
- Input/Output

# 4 Implementation

A compiler consists roughly of the following six phases: lexical analysis, syntactic analysis or parsing, semantic analysis, intermediate code generation, code optimization, and code generation.

## 4.1 Lexer and Parser

The lexer and the parser will be generated using BNFC, as it provides an ergonomic way of quickly developing and making changes to the grammar. If it turns out that the limitations of BNFC are too great, a custom lexer and parser will be created, perhaps using tools such as Alex and Happy.

## 4.2 Type checker

In the type checker, the abstract syntax tree from parsing is annotated with type information. This part of the compiler also validates the tree using the type-checking and type inference rules of System F. As validation is done in the type checker a lot of the syntactically valid programs from parsing are deemed invalid, thus only a small subset of programs can pass this step. Because of this, the next step of the compiler becomes a lot cleaner, as no validation is needed.

## 4.3 Code generation & Compilation

In these steps, the resulting annotated AST from the type checker is sent to the code generation function which traverses the tree, creating a new tree which represents LLVM's intermediate language, LLVM IR.

During this process, additional functionality, like writing text to the user, is baked into the program, as this is something that can not be implemented in our language due to the required bindings which are out of scope for this project. This finalized tree can be compiled into machine code using LLVM, which can then be run on the target platform.

## 4.4 Garbage collection

A garbage collector will be developed together with the language run-time and will be implemented in a lower-level language such as C++ to leverage its speed and lower-level operations. The implementation design is yet to be determined, as the project and tool chain structure are quite complex. One idea is to construct the garbage collector as an interface and invoke endpoints in LLVM as it supports external functions which do not have to be available at compilation. The benefit of this idea is that the garbage collector does not have to be recompiled every time the user compiles their program. However, this makes it such that the garbage collector has to be target specific, which results in more work for the developer to choose what platforms to support.

There are multiple algorithms for implementing a garbage collector and an endless amount of combinations. The algorithm used for this project will be determined later during the project when we are more confident in what suits the language's and the group's needs.

## 4.5 Documentation

As the language evolves the importance of the creation of a language specification and language reference grows with it. As with any proper programming language, having documentation on what the language includes and its capabilities is crucial for curious developers wanting to learn the language, or other developers wanting to create their own compiler for the language.

The specification will include the syntax and the grammar for the language, removing any ambiguity about what is a syntactically correct program. It should also include a description of the type system and some information about the compiler and the outputted machine code.

The language reference should act as a manual for the language, showing how to use the language and the basic elements such as the operators and the limited standard library.

## 4.6 Correctness

One important part of creating a programming language is verifying its correctness. The semantics should be sound and follow mathematical laws, and the behavior of the compiled code should be consistent and correct. Proving the correctness of the compiler is a complicated and time-consuming task. The focus will instead be properly testing all parts of the compiler. This will be done using unit tests in Haskell, as well as writing property-based tests using QuickCheck. [6]

## 4.7 Milestones & success

The success of the project is determined by how many of the listed milestones are achieved. The baseline of the project is covered by the first four points. Pattern matching and recursive algebraic data types are extensions to the language that make it actually practical to use. If time allows, garbage collection and linear types are more advanced concepts that will be explored.

- Type checking
- Type inference
- Code generation to LLVM IR
- Code generation to machine code using LLVM bindings
- Pattern matching

- Recursive algebraic data types
- Linear types
- Garbage collection

# 5 Societal and ethical aspects

Creating a programming language is generally not a question of ethics. The aspects to consider are mostly plagiarism and transparency. In this project, we will adhere to the code of ethics as defined by the Association of Computing Machinery [7]. The most important point in the code of ethics is 1.5 *Respect the work required to produce new ideas, inventions, creative works, and computing artifacts*, which states that original authors should be credited for their work, as well as respecting licenses and copyrights.

One should also follow point 1.3 *Be honest and trustworthy*, which states that one should be honest about the limits and capabilities of one's work, and our ability as well. This means we should be honest about flaws and untested parts of our project, and not exaggerate the capabilities of our language.

# 6 Timetable

| **Date** | **Activity** | **Contact** |
|---|---|---|
| 13/2 | Start working on the compiler | - |
| 7/3 | Half time presentation | - |
| 12/3 | First individual contribution report | - |
| 27/4 | Hand in the first draft of the report | Supervisor |
| 12/5 | Finalize the compiler | - |
| 15/5 | Hand in the second draft of the report | - |
| 15/5 | Contribution report | - |
| 17/5 | Hand in video presentation | - |
| 22/5 | Individual written opposition | Opposition group |
| 25/5 | Presentation | - |
| 26/5 | Presentation | - |
| 29/5 | Second individual contribution report | - |
| 2/6 | Hand in final report | - |

# References


[1] *The BNF converter*. [Online]. Available: http://bnfc.digitalgrammars.com/.

[2] *The LLVM Compiler Infrastructure*. [Online]. Available: https://llvm.org/.

[3] T. R. Foundation, *Influences - The Rust References*. [Online]. Available: https://doc.rust-lang.org/reference/influences.html (visited on 02/07/2023).

[4] J.-Y. Girard, "The system F of variable types, fifteen years later," *Theoretical computer science*, vol. 45, pp. 159–192, 1986.

[5] R. Griesemer *et al.*, *Hey! Ho! Let's Go!* Google Open Source, 2009. [Online]. Available: https://opensource.googleblog.com/2009/11/hey-ho-lets-go.html (visited on 02/06/2023).

[6] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of Haskell programs," in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, 2000, pp. 268–279.

[7] A. Association for Computing Machinery, *The code affirms an obligation of computing professionals to use their skills for the benefit of society.* 2018. [Online]. Available: https://www.acm.org/code-of-ethics.