# CS4201 Practical 2

160001055

October 2019

## 1 Introduction

THis practical was involved with implementing a full back to back compiler for the *Oreo* programming language. Using your language of choice, which in my case ended being Rust, the practical aimed to build on top of the parser that we delivered onto last practical in order to deliver the backend of the compiler, complete with variable resolution, scoping, type resolution and code generation.

## 2 Language Design

The entirety of the practical revolves on the choices made in the language design, and as such I believe it be important to discuss it first. As everything in computer science, Language Design involves many tradeoffs and pitfalls, and those can come to entirely shape the future of the language [1].

### 2.1 Structure

The program structure is Python like, with a main program top scope, on which functions are declared. This allows for easy examples, and less boilerplate for simple scripts. In case a user wants to revert back to a more C-like structure, that is always possible by defining a main function to use. Functions can be declared wherever a statement is accepted.

### 2.2 Scoping

Scoping is hierarchical with a couple of twists. First of all, each `begin`...`end` pair introduces a new scope, which can access each enclosing scope. The exception is with functions, which are pure and cannot refer to any variables in the enclosing scopes. Furthermore, each function is scoped to the scope which it is declared, so that utilities can be defined everywhere. In *Oreo*, variables can shadow each other, so for example a variable `i` can be declared multiple times in enclosing scope, and each reference to it will refer to the innermost declaration.

[1] Looking at you Visual Basic

### 2.3 Types

In *Oreo* there are three basic types: integer, boolean and string. On top of this we have function types that take in $n$ parameters of basic types and return one basic argument. Note that this disallows higher order functions, which seemed a bit too complex for this practical. The types are locally inferred for variables, and declared for function arguments. The inference rules for the variables allow for expressive programming that does not over rely on type declaration [2]. While in theory it would have been possible to infer function arguments as well, either by monomorphization at the call site or duck typing, or by inferring them by the usage, each of these strategies had some downfall that led me to move towards the simpler approach. First of all, monomorphization would have created more code duplication, longer compile times and deferred the type inference to the call site, which would have then required a more sophisticated type representation. Duck typing would have required to track type information at runtime, and for a program that is meant to be compiled down to HLA it would have been overkill to ship a runtime as well with each executable. Finally inferring by usage would have create some problems with, for example, functions that accept all data types such as `print`. Furthermore, changes in the inner body of a function can theoretically reflect on the function signature, which can then break the program in a non trivial way, and violate encapsulation. All things considered, allowing a simple type signature specification seemed like the best and most familiar option for the user.

## 3 Design

### 3.1 Parser

First of all, the practical involved a major rewrite of the parser, to both introduce new elements to the grammar (such as type information) and allow for better error reporting. In the previous design, the syntax tree was completely abstract, with no text information directly tracked. In practice this meant that any syntax item was not aware of their position in the source. Furthermore, the previous parser would halt on the first error, which makes the error reporting and fixing experience pretty bad. In order to fix these issues, the parser and the abstract syntax tree use a two layer representation. The bottom layer is an untyped syntax tree, which keeps track of the location of the item in source. This untyped layer can then be converted to a typed one, which (using the node database) offers more expressiveness and better edge case handling. Error handling is handled by returning an error node and continuing parsing as if the error did not happen. This allows for a fine grained and precise error reporting that for common mistakes, such as semicolons, can often parse the rest of program without hitches.

[2] Looking at you Java this time

## 3.2 Node Database

The node database is a solution that allows the typed layer to keep duplicated information and allocation to a minimum. It stores one single node, typically the full program node, and it assigns an unique identifier to each of the (recursive) children of said node. This allows to completely abstract away the full untyped node representation and just operate with the typed syntax tree, calling to the node database for error handling and similar.

## 3.3 Scoping

The first operation is handling scoping. The way this is done is by building up a hierarchical lists of scopes, together with the variable declarations in each of these scopes. This is achieved by a in order depth first tree traversal, which includes a new scope every time a new compound statement is encountered. Some subleties in this have to be mentioned. Every time a new identifier is encountered, they are assigned a unique identifier, that is then used in every of the following stages. Furthermore, each identifier is associated with a reference to their declaration context, which is then later used to resolve types. There is a distinction here between function scopes and normal scopes, which is needed to disallow accidental closures.

## 3.4 Resolution

The next step is scoping resolution. This involves a similar walk of the syntax tree, only that on each identifier node encountered (that are all conveniently all collapsed to the same type) it will look up in the scope tree in order to figure out what identifier it refers to. Differently from the scoping section, this operation is fallible, and, similarly to the parse tree, we allow for detecting all resolution errors at once.

## 3.5 Type Inference

The next step, and possibly the most important one[3] is type inference and/or resolution. As in the previous two section, we walk the abstract syntax tree, type checking every expression that we find around the way. The typing rules are as follows:

$$\texttt{true} : \mathit{bool} \land \texttt{false} : \mathit{bool}$$

$$\texttt{num} : \mathit{int}$$

$$\texttt{str} : \mathit{str}$$

$$\forall T \in \{\mathit{int}, \mathit{bool}, \mathit{str}\} : \frac{\Gamma \vdash x : T}{\Gamma \vdash y : T}(\texttt{var}\ \ y \coloneqq x)$$

[3] In my mind, even tough full disclaimer I am a bit of a type junkie (https://aphyr.com/posts/342-typing-the-technical-interview for a good read)

$$\frac{\Gamma \vdash x : int, y : int}{\Gamma \vdash x \oplus y : int}$$

$$\frac{\Gamma \vdash x : int, y : int}{\Gamma \vdash x \leq_R y : bool}$$

$$\frac{\Gamma \vdash x : bool, y : bool}{\Gamma \vdash x \ \texttt{op} \ y : bool}$$

$$\frac{\Gamma \vdash x : bool}{\Gamma \vdash \texttt{not} \ x : bool}$$

$$\frac{\Gamma \vdash f : T_1 \times \cdots \times T_n \to T_0, \forall i : x_i : T_i}{\Gamma \vdash f(x_1, \ldots, x_n) : T_0}$$

Note that we use $\oplus$ to signify any of $\{+, -, *, /\}$, $\leq_R$ for any of $\{=, \leq, \geq, <, >\}$ and `op` for one of $\{\texttt{and}, \texttt{or}\}$. Also, we don't show how the derive a function declaration from the environment, but that is done by reading the signature. The typing module uses this rules to deduce types for expression, associates types to identifiers based on their declaration and finally checks all of the above for consistency in the whole program. As the above modules, it reports the error on a rolling basis, so it is able to discover multiple typing errors at once, and report them to the user accordingly. Just for fun, in the above sections I mentioned closures quite a lot. That is because I had them implemented at all those levels until typing, and the scheme above was able to correctly resolve everything, and even correctly flag uninitialized closures arguments, unfortunately I couldn't figure out how to move them down to TAC and HLA which was disappointing.

### 3.6 TAC

Translation to TAC is relatively straightforward, thanks to a host of temporary variables we conjure out of the void. Confident of the fact that the before phases will have ironed out every invalid program, we once again (and finally) traverse the abstract syntax tree. Thankfully our grammar already completely disambiguates precedence, so each expression can be transformed by introducing at most one temporary. Loops and ifs are easily translated thanks to a combination of conditional and unconditional jumps, and functions calls are kept at an high level format, trusting HLA for the manual popping.

### 3.7 HLA

Finally the HLA sections is a quite straightforward if not tedious transformation from the TAC code. Almost every operation is transformed as is, with some care taken in order to load and track variables in registers and the similar. Each variable that appears in the source is stored on the stack, while temporaries are exclusively allocated to registers (as they are more likely to be used next). While I was not able to test HLA (as I did not have access to the lab for a lot of the time), I am confident that at least the main logic should be working.

# 4 Implementation

This section just highlights some of the sections of the practical, that seemed to me more interesting.

## 4.1 Parser

The parser uses a pattern that I have grown more and more fond of. It creates a builder object that consumes the input stream. This builder object keeps track of the state and keeps track of the position of the node in the input stream. We can operate on this builder object by then making recursive children calls, and change the type of the node to be built by peeking one step ahead (always possible since the grammar is LL(1)).

## 4.2 Node Database

One of the most peculiar implementation trick that I used is in the node database module[4]. In order to discuss this a small detour is needed. Rust uses a memory model that relies on lifetimes, and in particular makes sure that if a value has a reference pointing to it ("is borrowed" in the lingo), then it has to live and not move for at least as long as the reference. This works really well for a lot of cases, however sometimes it is insufficient. In our case, in order to save memory, we aim to store reference to the children of the node stored in the database, and associate each of these with an id. However, this does not play well with the borrowing rules, as if both the references and the value that they refer to are in the same struct, then the references will become invalid if the struct ever is moved. Luckily, this is a well know problem in Rust, know as self-referential struct problem[5], and it has a solution, namely using the `Pin` API to statically promise to the compiler that the struct will not ever be moved. While describing it in words is kind of complicated, I would suggest the reader has a look as it is quite fascinating.

## 4.3 TAC

In both the TAC and HLA section, I made it so that instead of directly building the output the program goes trough an intermediate typed state, which allows for greater safety and more confidence, at the cost of considerable more verbosity. In particular, this also would allow for different output based on external factors such as target architecture or similar.

[4] and my first time having to use `unsafe` in Rust!

[5] https://boats.gitlab.io/blog/post/2018-01-25-async-i-self-referential-structs/ for an introduction. Self referential structs have been for a while one of the big blockers in the async-await push

### 4.4 HLA

In order to save some registers, the HLA translation does some interesting optimizations which I think could be interesting. For example, for conditional jump instruction that takes in a constant operand, we automatically replace this with the corresponding unconditional jump, saving one memory load and any more setting of the comparison registers. Relying on the fact that each temporary is ever read once, another optimization that we use is for setting a variable to the value of a temporary. Instead of executing a `mov` instruction, we just rename the variable to the temporary in the representation, which allows for some good performance savings.

## 5 Testing

As in the previous practical, testing was conducted with the use of snapshots. We generally aim to have each test in a module to follow a similar structure. The program is parsed directly from a string, and then the result is compared with previous results, to check that breakage did not happen. I have around 90 tests, each focusing on one particular edge case of the program, and which also double as documentation to what the output of the program might be.

## 6 Documentation

While in the last practical I completely forgot that comments and documentation existed, in this one I made a conscious effort to make everything as clear as possible. In particular, I used `#[deny(missing_docs)]` to ensure that every public item is correctly documented (and the code won't compile if that doesn't hold), furthermore I tried to comment more critical code sections, so hopefully that helps reading the code a bit. Running `cargo doc --open` will also open a browser windows with all documented item, which should help navigating the code.

## 7 Conclusion

In conclusion, this practical was a challenging yet entertaining one, during which I got the chance to experiment and develop a back to back compiler, which included parsing, scoping, name resolution, type resolution, three address code generation and HLA. I believe that the entire process was enlightening for many reasons. First of all, it shows how even a simple language like *Oreo* originate complex design design and complex compilation system. This puts in perspective how impressive and complex the design for more complicated programming languages is, and why so often bugs in compilation occur [6]. In conclusion, I

[6] This time I am thinking about the MSVC C++ compiler two phase name lookup bug which bricked a lot of my metaprogramming programs

really and wholesomely had fun in this practical[7], and I hope to continue with programming language design in the future.

[7]Save for the HLA part, that was quite painful, next time I will let LLVM handle it for me