# Rix

## Client goals, starting condition

The Rix Compiler is a project that was undertaken as a student at BCIT in the fall of 2015. We were a team of three students and the project came to us from an external client who was himself a developer.

Statically typed,

readable and easy to use like Python

speed and power, low-level access of C

Type inference,

? operator

Functions,

Classes with single inheritance

Whitespace sensitivity.

Started with: large single file, 300+ line functions, some functions and control-flow statements. Many hard-coded aspects.

Started to understand the existing codebase by documenting. Tested to identify bugs and understand what’s going on.

## Learning about compilers

None of us knew about compilers.

Read up on how they worked,

Compilers are generally split into multiple processes, generally including:

lexical analysis, parsing, building a syntax tree, generating code in the output language.

For this project we would have to touch all of those areas, and also manage the scope and symbol table, and type-checking. The output language is C code, so no assembly or machine code was needed, and we didn’t touch optimization in any way.

## Lexical analysis

Lexical analysis (lexing) is the process of turning a string of characters into tokens. Basically you crawl through the text files character by character and when your string of characters makes a complete token you pass it forward to the next compiler phase. An English comparison would be connecting the letters of this sentence into words, broken up by spaces and punctuated with periods and commas. A Rix example might be:

n = 5

i for 1, n+1

print i

would get broken into

n = 5 \n i for 1 , n + 1 \n \t print i

The lexer has some vague idea what these tokens mean; It recognizes that = is the equality operator, and “print” is some identifier for a function or variable. However, the lexer has no idea which tokens relate to which others. It just converts the string of characters into a string of tokens for the next process: parsing.

## Parsing

Parsing takes the tokens from the lexer and starts connecting them into combinations that start to have some meaning. It tries to match sequences of tokens against a set of patterns it recognizes. For example, in English you often form sentences out of a sequence of tokens like subject, verb, object, period. It’s not the only way to build a sentence but it’s one of the patterns the parser would match.

In Rix, the parser tries to form statements in a similar subject-verb-object format, followed by a newline.

n, =, 5, \n would be parsed as a complete statement in itself, as would

i for 1, n+1

where the object itself has a nested subject-verb-object.

## Flex and Bison and EBNF

Doing all this string processing and pattern matching could have been a massive endeavour in itself but this project also happens to be released under an open source licence and we could employ the Flex and Bison tools for lexing and parsing.

Flex lets us set up regular expressions on characters (and some more complicated rules) to define each of the tokens, and Bison let us set up something like regular expressions for tokens to let us define recursive patterns.

Here’s where a language definition technique called EBNF (Extended Backus-Naur Format) comes into play. You define a symbol that represents any program, and then what that program can be made up of, similar to XML DTDs. This repeats recursively until you’re down to symbols that are made up of the tokens that the lexer is spitting out.

A simplified EBNF for Rix might look something like this:

* Program: statements+, EOF
* Statements: class | function | statement | indent statement unindent | EOL
* Statement: subject?, verb, object (comma object)\*, EOL
* Subject: integer | float | variable | …
* Verb: function | plus | minus | equals | ….
* Object: integer | float | variable | statement ….
* Function: functionHeader, indent, statement+, unindent
* Class: classHeader, indent, state\*, function\*, unindent

This is really cool! Once you have a dictionary like this of what composes what else, you can directly build a tree out of your program! Remember that Rix code above?

n = 5

i for 1, n+1

print i

It deconstructs into something like this:

Program

Statements+, EOF

Statement statement indent statement unindent

Subject verb object; subject verb object comma object; verb object

Once the parser has processed all the tokens into Rix syntax, it’s ready to test for errors (like: is this variable defined in this scope?) and generate the code for output.

## Feature set added

Variables declared, expressions evaluated

Functions defined, callable. Work with 0, 1, and more arguments

Conditionals were handled with a ? operator that tracks the result of the previous operation

Flex and Bison were integrated to handle lexing and parsing fully

Classes worked with class members, methods and constructors

Classes inherit members from parent classes, and you can access those parent members.

You could instantiate classes, and access members and parent members, even inside of functions, or call functions on classes.

## Creation and Design work is exciting

This project was a serious stretch of brain power for me and it was the most exciting project I’d worked on to that date! From learning and understanding the new topic of compiler architecture, to designing and implementing a solution to tracking scope and symbols, to figuring out a way to implement object-oriented programming in C it was an amazing experience. The compiler is far from complete, and has continued to evolve since I completed my work on it. Some of the next challenges are implementing generics, building up a standard library, and opening the language up for redefining all the built-in symbols so that Rix’s default dialect (behaviour) will be written in Rix itself and open to reimagining.

Check out the project on GitHub!