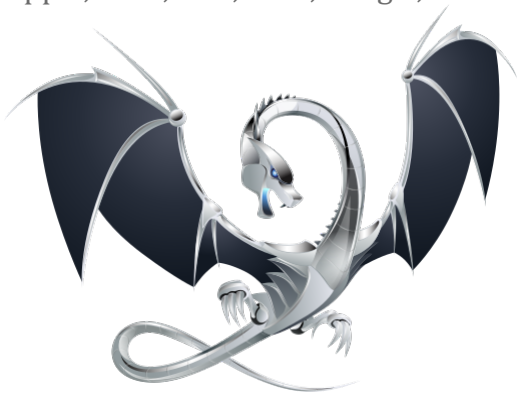
LLVM Everywhere

Some of the most recent features added to Xojo, including iOS, 64-bit apps, and Raspberry Pi have been made possible by LLVM. Read on to learn more about it.

What is LLVM?

On a high-level, LLVM is a collection of libraries and tools for building compilers. The LLVM project was started in the early 2000s by Chris Lattner (who eventually went to work at Apple and has now moved on to Tesla) and remains in active development. Because LLVM is a toolkit for building compilers and because it can target most CPU architectures, its use has become much more prominent in recent years. It has widespread industry backing with support from major companies including AMD, Apple, ARM, IBM, Intel, Google, Mozilla, Nvidia, Qualcomm, Sony, Samsung.



The first appearance of LLVM in Xojo was when we started using it to compile and run XojoScript code (2010). Later we used LLVM to build iOS apps (2014), 64-bit apps for x86 (Windows, MacOS, Linux) and 32-bit apps for Raspberry Pi (2015), and most recently we've hooked up the Xojo debugger, so it works with apps built using LLVM (2016).

In addition to now being used as part of Xojo's compiler, LLVM is also used with other languages, including: Clang, D, Rust, OpenCL, and Swift.

Why is Xojo using LLVM?

First, a little background. A compiler consists of many things and is typically split into parts called the "front end" and the "back end". LLVM has components for both front ends and back ends; Xojo uses it as a back end.

For 32-bit x86 apps, Xojo uses its own in-house, proprietary compiler first created in 2004/2005. This powerful and fast compiler handles the front end and back end, but it does have a couple limitations: it can only target 32-bit x86 and it does not do any optimizations.

But technology marches on and we knew we would eventually have to add support for x86-64 (64-bit) and ARM CPU architectures. Updating a compiler is a large undertaking, but fortunately the LLVM compiler toolkit started gaining traction in the industry and became a better alternative to creating our own x86-64 and ARM compiler back ends.

Today when you build an iOS app, a 64-bit app for Windows, MacOS or Linux, or an ARM app for Raspberry Pi you are using LLVM as the back end to generate your native, binary code.

By using LLVM, we are able to respond more quickly to changes in the industry and to add features our users want. As an example, we were able to rapidly add Raspberry Pi support (32-bit ARM Linux) by leveraging much LLVM work that had already been done for x86-64 and ARM for iOS.

As Joe Ranieri (the Xojo compiler guru) likes to say, "Every single line of code [you write] is a liability and not an asset". With LLVM we can implement more features in less time because we don't have to recreate the wheel, so to speak.

In effect, LLVM enables us to better support the unknown future, including new and updated architectures. One such example is 64-bit ARM Linux, which will likely become necessary at some point.

Learn More

LLVM is great. We love it and are thrilled that it has allowed us to add significant capabilities to Xojo for you. A compiler is a complicated thing and we are pleased that Xojo hides the complexity of compiler technology so that you don't have to worry about it. You just need to focus on making the best app you can, select your OS target, click build and let Xojo take care of the rest.

If this post has made you curious about LLVM and you'd like to learn more, I recommend the following:

- Wikipedia: LLVM
- LLVM official site
- Accidental Tech Podcast: Interview with Chris Lattner, creator of LLVM
- Joe Ranieri's Compiler Session from XDC 2016

Overview and Lexer

At XDC 2016 there was a lot of interest in Joe Ranieri's Compiler session where he talked about compilers and LLVM. I've already summarized a bit about LLVM in an earlier post, but after talking with Joe we decided to put together a series of blog posts on compilers.

These will all be at a high-level. None of these posts are going to teach you how to write a compiler. The goal of these posts is for you to have a basic understanding of the components of a compiler and how they all work together to create a native app.

Compiler Components

A compiler is a complicated thing and consists of many components. In general, the compiler is divided into two major parts: the front end and the back end. In turn, those two parts have their own components.

For the purposes of these posts, this is how we will be covering the components of the compiler:

Front End

The front end is responsible for taking the source code and converting it to a format that the back end can then use to generate binary code that can run on the target CPU architecture. The front end has these components:

- Lexer
- Parser
- Semantic Analyzer
- IR (intermediate representation) Generator

Back End

The back end takes the IR, optionally optimizes it and then generates a binary (machine code) file that can be run on the target CPU architecture. These are the components of the back end:

- Optimizer
- Code Generation
- Linker

Each of these steps processes things to get it a little further along for the next step to handle.

The linker is not technically part of the compiler but is often considered part of the compile process.

Lexer

The lexer turns source code into a stream of tokens. This term is actually a shortened version of "lexical analysis". A token is essentially a representation of each item in the code at a simple level.

By way of example, here is a line of source code that does a simple calculation:

```
sum = 3.14 + 2 * 4
```

To see how a lexer works, let's walk through how it would tokenize the above calculation, scanning it from left-to-right and tracking its type, value and position in the source code (which helps with more precise reporting of errors):

1. The first token it finds is "sum"
    1. type: identifier
    2. value: sum
    3. start: 0
    4. length: 3
2. Token: =
    1. type: equals or assigns
    2. value: n/a
    3. start: 4
    4. length: 1
3. Token: 3.14
    1. type: double
    2. value: 3.14
    3. start: 6
    4. length: 4
4. Token: +
    1. type: plus
    2. value: n/a
    3. start: 11
    4. length: 1
5. Token: 2
    1. type: integer
    2. value: 2
    3. start: 15
    4. length: 1
6. Token: *
    1. type: multiply
    2. value: n/a
    3. start: 15
    4. length: 1
7. Token: 4
    1. type: integer
    2. value: 4
    3. start: 17

4. length: 1

As you can see, white space and comments are ignored. So after processing that single line of code there are 7 tokens that are handed off to the next part of the compiler, which is the Parser.

Parser

At XDC 2016 there was a lot of interest in Joe Ranieri's Compiler session where he talked about compilers and LLVM. After talking with Joe, we decided to put together a series of blog posts on compilers. These are at a high-level. None of these posts are going to teach you how to write a compiler. The goal of these posts is for you to have a basic understanding of the components of a compiler and how they all work together to create a native app.

After the Lexer has converted your source code to tokens, it sends them to the Parser. The job of the Parser is to turn these tokens into abstract syntax trees, which are representations of the source code and its meaning.

For reference, this is the simple source code we are "compiling" as we go through the parts of the compiler:

```
sum = 3.14 + 2 * 4
```

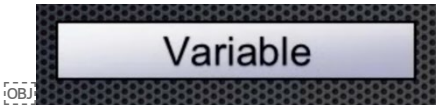
The lexer has converted this to a stream of tokens which are now sent to the Parser to process. The tokens are:

1. Type: identifier
    1. value: sum
    2. start: 0
    3. length: 3
2. Type: equals or assigns
    1. value: n/a
    2. start: 4
    3. length: 1
3. Type: double
    1. value: 3.14
    2. start: 6
    3. length: 4
4. Type: plus
    1. value: n/a
    2. start: 11
    3. length: 1
5. Type: integer
    1. value: 2
    2. start: 15
    3. length: 1
6. Type: multiply
    1. value: n/a
    2. start: 15
    3. length: 1
7. Type: integer
    1. value: 4
    2. start: 17
    3. length: 1

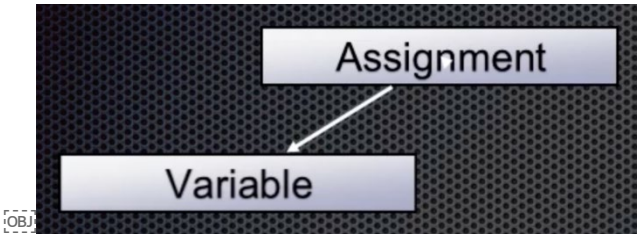
Parser

To see how this works, we'll go through the tokens and create the syntax tree.

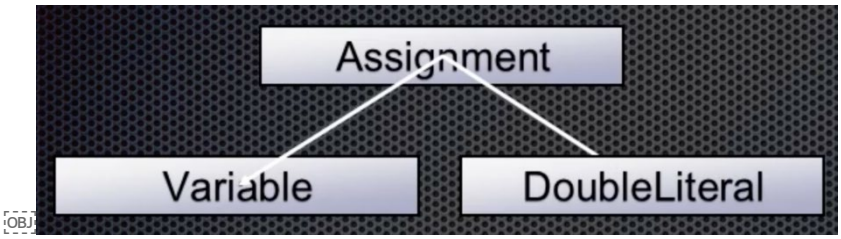
The first token is the identifier, which the parser knows is actually a variable. So it becomes the first node of the tree:



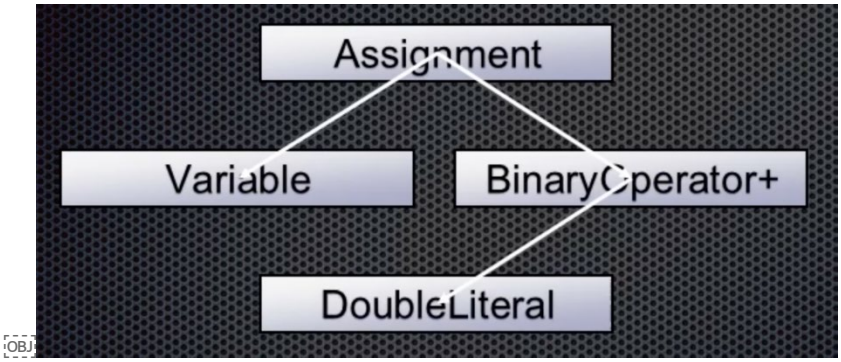
The next token is equals or assigns. The parser knows things about this such as that it is an assignment and that assignment is a binary operator that has two operands, one on the left and one on the right. The variable from above is the left value, so it gets moved to the left side of the Assignment node that is now added to the tree to look like this:



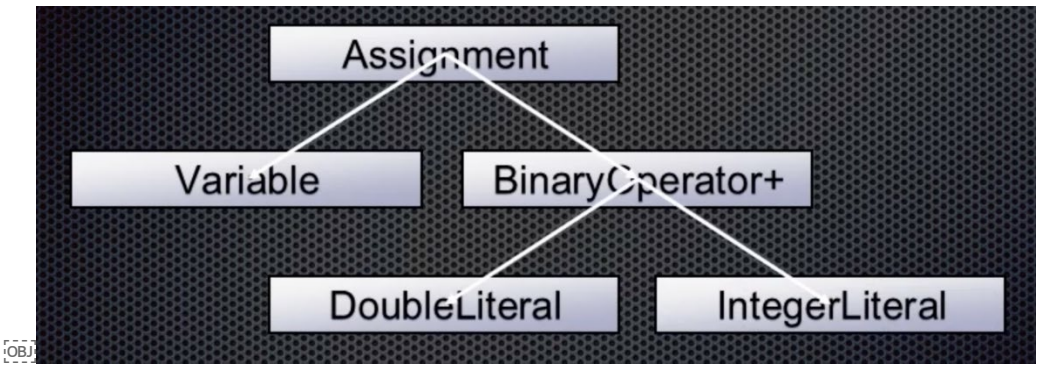
Continuing, the double token is next with value 3.14. This is the right value for the assignment:



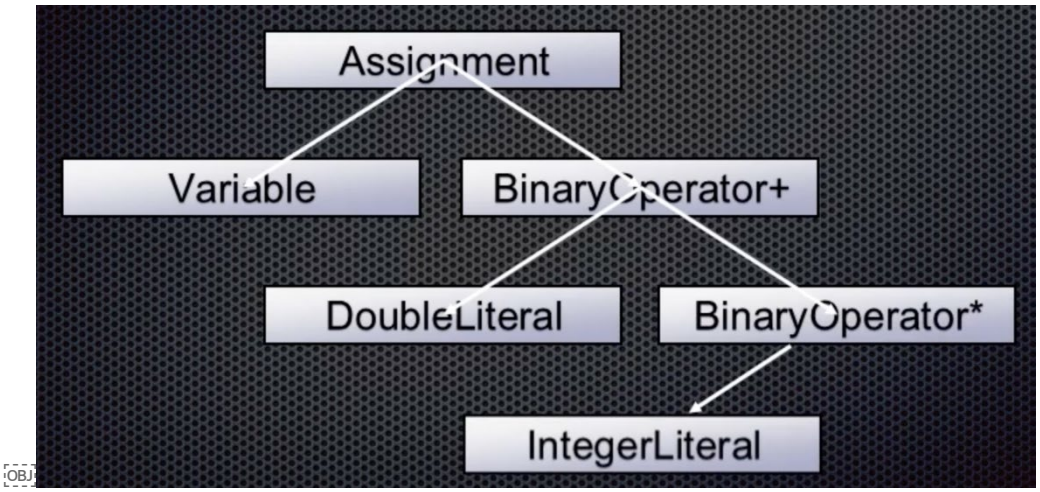
Moving along, the plus token is next. The parser knows this is the addition operator (BinaryOperator+) that takes two values (and is also left associative). This means that the addition is added to the tree with the double as its left value:



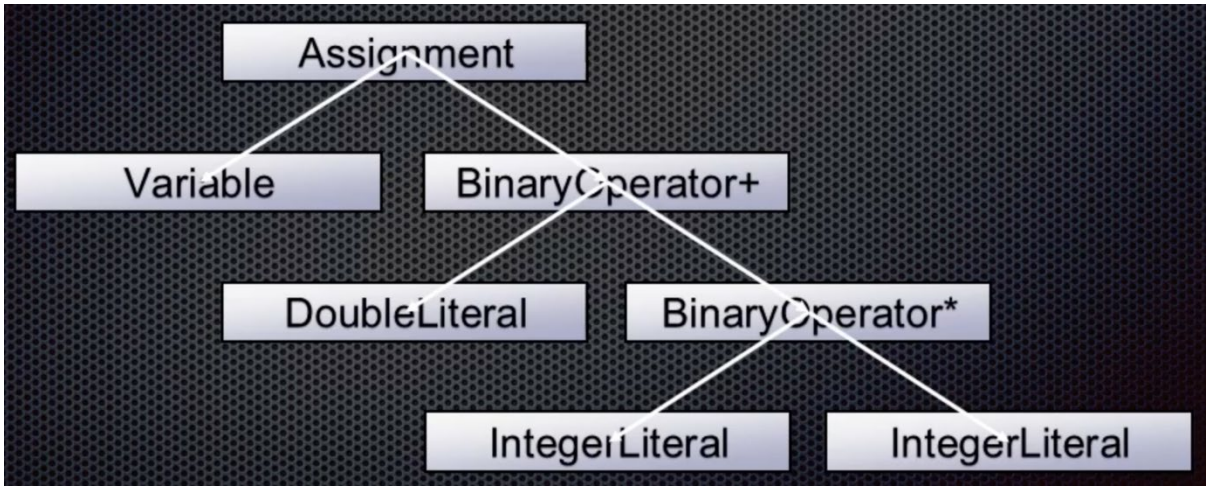
After the plus token, an integer is next, and this becomes the right value for the BinaryOperator+ node:



Next is the multiply token, another binary operator that is left associative. So it gets the integer as its left value:



The last token is another integer which becomes the right value for the multiplication:



And that is the final abstract syntax tree for our simple line of code. The Parser has done its work and has now created a tree that no longer represents the exact source code but is an idealized representation of what the user wrote.

This tree is then provided to the next component, the Semantic Analyzer.

Semantic Analyzer

The Semantic Analyzer is the real heart of the compiler. Its job is to validate code and figure out what the code means. Essentially it validates that the code is semantically correct.

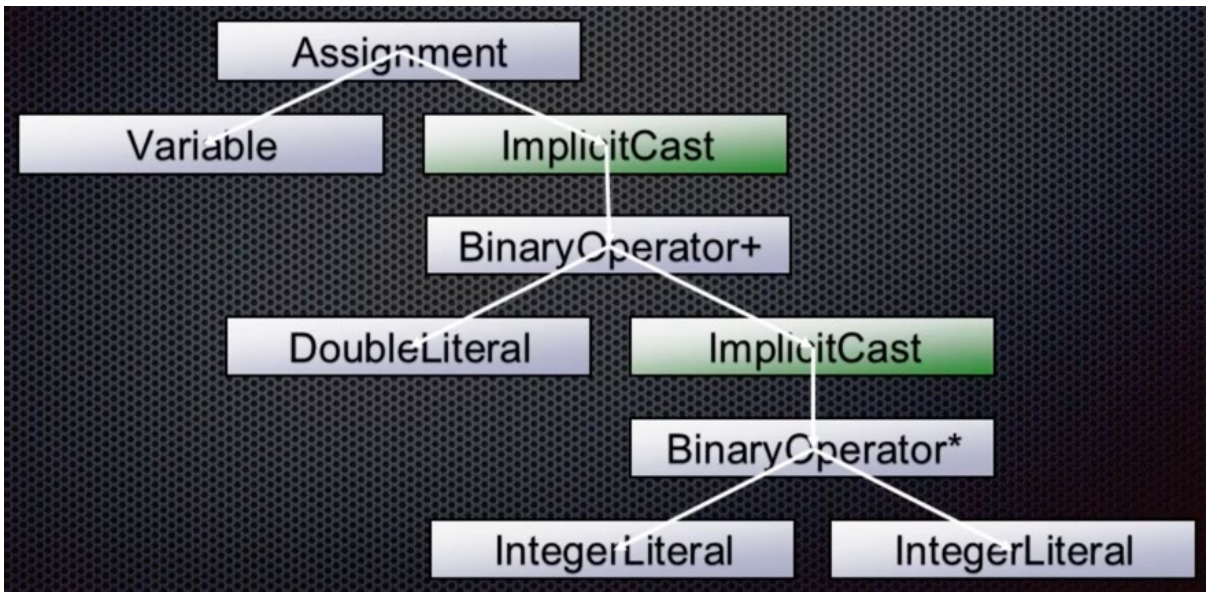
Semantic Analyzer

With the output from the Parser, all the compiler has is what the user actually typed, although converted to a format that is easy for the compiler to digest.

The Semantic Analyzer knows all the rules regarding the programming language. For example, it knows that an Integer can be multiplied with a Double. It knows a String cannot be compared with a Double. It knows how to do assignments to variables. It knows that Objects can be used with the “New” command. It knows about scope information. Everything that the compiler knows to pass the initial syntax check is in the Semantic Analyzer.

If we start with the syntax tree produced by the Parser, the Semantic Analyzer will go through it and add information to the syntax tree regarding types. And not just types but adds information about things that are implicit in the language. For example, you may not think about it but there are implicit conversions that happen should you do something like assign a Double to an Integer.

As you can see below, the Semantic Analyzer has gone through the syntax tree and updated it to include when implicit conversions (ImplicitCast) are done.



This updated tree is then handed off to the next part of the compiler, IR Generation.

IR Generation

Now that the Semantic Analyzer has verified that the code is correct and created syntax trees, it's time to talk about IR generation.

What is IR?

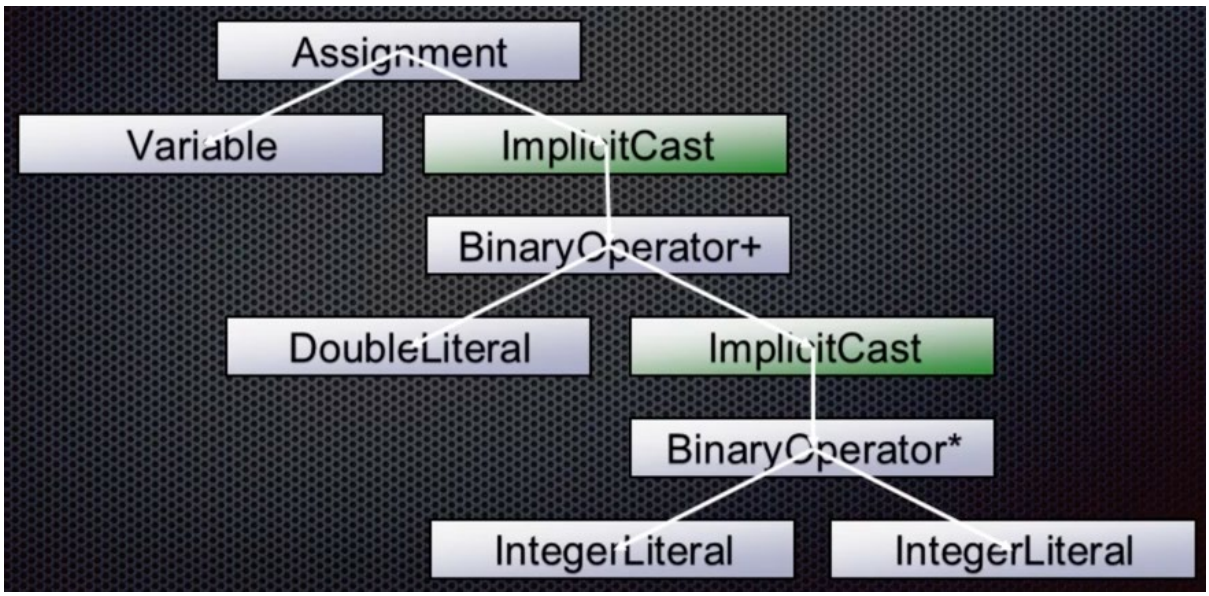
After the Semantic Analyzer, the next step is to turn the trees that it validated and added type information to into a representation that is much closer to what the machine is going to generate. This representation is called an intermediate representation (IR). The Xojo compiler, when building for 64-bit or ARM, uses LLVM IR.

The resulting LLVM IR describes the actual control flow of the program and every single thing that will be in the final program. In addition to the user-visible code that gets executed and the implicit conversions that are now in the trees, it also needs to contain all of the hidden, behind the scenes calls like reference counting and introspection metadata.

LLVM IR is actually higher level and more abstract than the actual assembly that it'll end up generating. For example, unlike assembly language, it's entirely strongly typed and the Xojo compiler has to be very precise in what it generates (which is a good thing!).

IR Code Generation

To get started, here is the abstract syntax tree that was previously created by the Semantic Analyzer:



To generate IR, the compiler walks through the above tree, depth first, to get to the leaf nodes. Doing this gets us to the BinaryOperator* for the multiple on the lower right side of the tree.

The LLVM IR to multiply those two values (mul) looks like this:

```
%1 = mul i32 2, 4
```

Now it works backwards through the tree. So, the next item is the implicit cast, which has to cast the value that was calculated in the previous command:

```
%2 = sitofp i32 %1 to double
```

The sitofp IR command means “Signed Integer to Floating Point”.

Continuing up the tree, the binary operator is next, so it can now grab the left hand-side value to apply to the right-hand side value. This is the IR to add the values:

```
%3 = fadd double 3.14, %2
```

The fadd IR command means “floating point add”.

And continuing up the tree, the implicit cast is next:

```
%4 = fptosi double %3 to i32
```

The fptosi IR command means “floating point to signed integer”.

Lastly, we reach the actual assignment (store) with IR that looks like this:

```
store i32 %4, i32* @sum
```

Here is the complete IR that gets generated:

```
%1 = mul i32 2, 4
%2 = sitofp i32 %1 to double
%3 = fadd double 3.14, %2
%4 = fptosi double %3 to i32
store i32 %4, i32* @sum
```

Reading through this you should now understand why no one wants to manually write code at such a low-level.

This is the last part of the compiler that is considered to be part of the front end. The rest of the compiler components belong to the back-end.

Back-End Overview

Once the front end has done its work it's time for the back-end components to take over.

Back-End

The components of the back end take the IR that was generated by the last step of the front end and emit executable code, which is machine language in the case of Xojo.

To recap a bit from the LLVM post, for 32-bit x86 apps, Xojo uses its own in-house, proprietary compiler first created in 2004/2005. This powerful and fast compiler handles the front end and back end, but it does have a couple limitations: it can only target 32-bit x86 and it does not do any optimizations.

Today when you build an iOS app, a 64-bit app for Windows, MacOS or Linux, or an ARM app for Raspberry Pi you are using LLVM as the back end to generate your native, binary code.

The rest of this book will cover the back-end as it pertains to LLVM. Specifically, the components are:

- Optimizer
- Loop Unrolling
- Code Generation
- Linker

Optimizer

An optimizer “improves” the IR, but that can mean a lot of different things. Improve could mean “run faster” or “use less memory”. Or perhaps you want to optimize for memory access time because CPUs are so fast it is sometimes more efficient to repeatedly calculate something rather than calculate it once, store it and access it later.

The Optimizer does a series of transformations to the IR code, typically in multiple passes. LLVM provides full control over these passes.

Not all LLVM optimizations provide benefits to Xojo code. We have distilled the many complicated optimization settings that are available with LLVM into three options that are useful for Xojo code, which you can set from the Shared Build settings: Default, Moderate and Aggressive.

The Default optimization does minimal optimization in order to have the quickest compile times.

The Moderate setting does more optimizations, primarily to reduce the time needed for mathematical calculations which results in slightly slower compile times.

The Aggressive setting does many more mathematical optimizations to further reduce calculation time, but also dramatically increases compile time.

Optimization intends to create something that is equivalent to the original code. The result is the same, even if the means to do so might be very different. For example, the optimizer may determine that it should do a bit shift to do integer math as a single operation rather than a series of add operations. This could result in smaller, faster code but may take longer for the optimizer to process the code to make this determination.

Deciding the "best" optimization for any code is not technically a solvable problem (np-hard), so optimizers use a combination of "heuristics and hand-waving". "Hand-waving" means the compiler thinks this is correct but has no real way to prove it. And heuristics simply means that optimizations that have been known to work in prior code are used when similar code is found.

Constant Folding

Constant Folding is a simple example of an optimization that can be done.

This essentially mans the optimizer evaluates constant expressions up front to reduce execution time, and save stack and register space.

For example:

```
a = 1 + 2
```

can be replaced with:

```
a = 3
```

Not everything will be quite so obvious in your code, of course.

Optimization Issues

There are certain types of code that can make the optimizer's job more difficult. These special Xojo features can challenge an optimizer:

- Exception Handling: Makes things difficult because you cannot tell where a function call may return.
- Memory Management: Xojo has deterministic object destruction when variable goes out of scope, which forces the optimizer to have to track things more closely.
- Introspection: The use of Introspection requires lots of metadata to remain available so it can be referenced at run-time.
- Threading: Cooperative thread yielding affects loops and other things.
- Xojo is a “safe” language and many of the things it does to ensure your app does not crash (and instead raise exceptions) such as Nil object checks, stack overflow checks, bounds checks, etc. all restrict what the optimizer can do.
- Plugins: Since these are pre-compiled, they are ignored by the optimizer.

Continue to learn about a specific optimization: Loop Unrolling.

Optimizer Loop Unrolling

The last post covered optimization in general. In this post you'll look at a specific optimization called "loop unrolling".

Loop Unrolling

Loops are a key optimization point. Unrolling a loop means that you repeat the code content of the loop multiple times. It is essentially exactly what you are taught not to do when writing code.

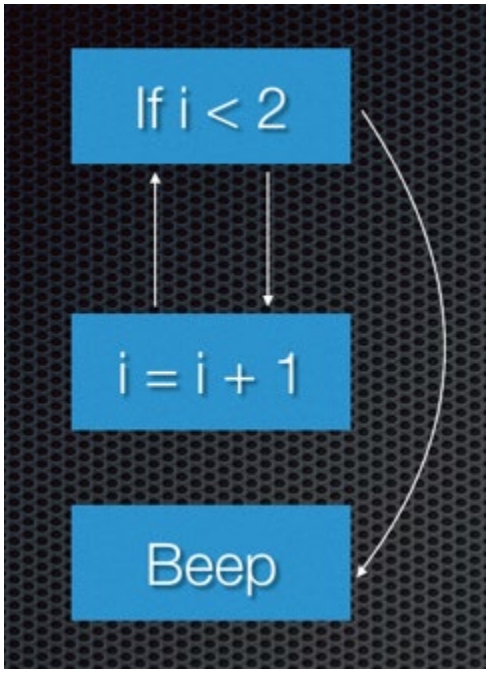
Loop unrolling avoids costly branches. This will not necessarily unroll the entire loop so that you get code repeated 100s of times, but it may unroll it a bit so the code repeats a few times.

Modern hardware hates branches because it makes other optimization more difficult, so unrolling a loop enables additional optimizations for both the compiler itself and the CPU.

Here is a simple loop:

```
Dim i As Integer = 0
While i < 2
  i = i + 1
Wend
Beep
```

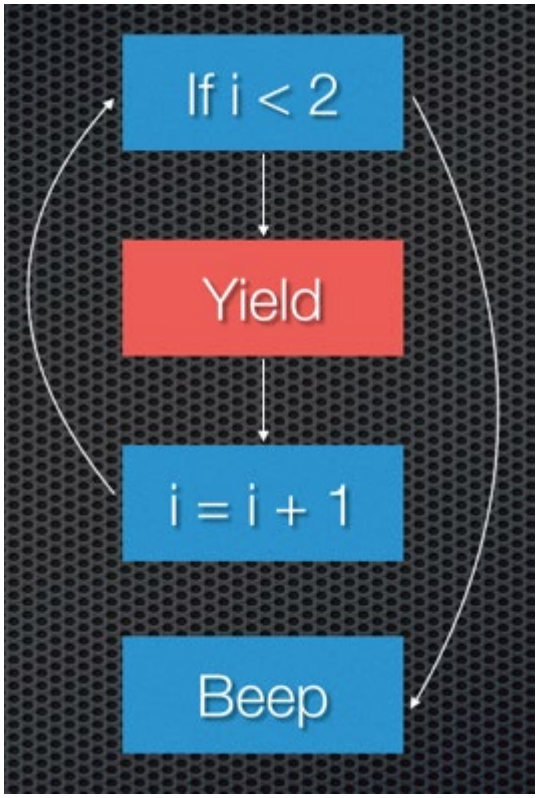
This gets modeled as a control flow graph:



A control flow graph is a directed graph.

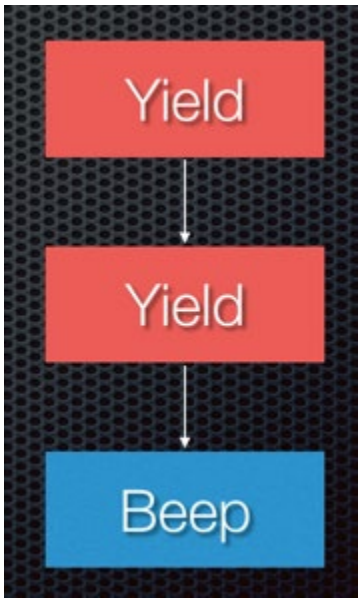
Looking at the above, the while gets converted to an if. After the integer assignment, there is a branch back to the if. If the condition is false, then it branches to the beep. This is essentially a Goto, for old-timers.

But the compiler may also insert stuff on your behalf. For example, Xojo adds Yield function calls to enable cooperative threading and these function calls cannot be removed by the optimization process.



So to unroll the loop, the idea is that there are only two iterations of the loop which the compiler is able to determine.

So unrolling results in this:



In this short and simple example, the value of the “i” variable is never used so it can be discarded, saving both iteration time and storage space.

With optimization complete, the compiler moves on to Code Generation.

Code Generation

Code generation is one of the last steps of the compiler. This is where the compiler emits actual machine code for the IR that was previously created.

This is the simple code we started with in the Compilers 101: Overview and Lexer

```
sum = 3.14 + 2 * 4 // calculation
```

It results in a constant value of 11. After the IR is generated and optimized, it can boil down to just a single line of machine code, which will vary by processor and architecture. Machine code is just binary and not readable, so below is what the Assembly code might look like.

This is the Assembly code for 32-bit ARM:

```
movs r0, #11
```

This is the Assembly code for ARM64:

```
movz w0, #11
```

x86 and x86-64 use this Assembly code:

```
movl $11, %eax
```

Obviously, this is the tricky part of making a multi-platform compiler since Assembly code is different between processors and architectures.

Once you have machine code that the computer can run, the last step is to link all the pieces together so that you have an app that the OS can run. This is done by the Linker.

Linking and Wrap-up

The linker is not technically part of the compiler, but it is needed to make a completed app. The purpose of the linker is to combine (link) all the various bits and pieces of machine code created by the compiler along with the necessary information to create a runnable app for the OS.

Each OS has a way to define an executable file so that it can run it. This typically involves some sort of header and a format that describes how the various machine code binary components are all combined. This is called an executable type, executable format or object file format.

This is what Xojo uses as the executable format when building 64-bit x86 and ARM apps using LLVM:

- For Linux, Xojo uses ELF (Executable and Linkable Format). This is a common standard used by many Unix-like systems.
- For macOS and iOS, Xojo uses Mach-O. This is the format that was introduced for Cocoa apps and dates all the way back to the NeXT days.
- For Windows, Xojo uses the PE (Portable Executable) format.

The Linker is responsible for combining all the machine code generated by the compiler for your project, adding libraries and generating the appropriate executable

format. When the linker has finished, you have a native app that works on the operating system.

For 64-bit x86 and ARM apps using LLVM, Xojo uses the lld linker.

Wrap-Up

I hope you found the Compiler Series helpful. And when you need to easily create your own cross-platform and multi-platform apps for Windows, macOS, Linux, Raspberry Pi, iOS or the web be sure to use Xojo!