A Report

On

NextLeap

Submitted to

University of Petroleum and Energy Studies

In Partial Fulfilment for the award of the degree of

BACHELORS IN TECHNOLOGY

In

COMPUTER SCIENCE AND ENGINEERING (with specialization in IOT & SC)

By

Arpit kansal

500091746

Under the guidance of

University of Petroleum and Energy Studies

Dehradun-India

November 2023

Table of Contents

MY COMPILER BASICS

Operator precedence

When parsing the expression `1 + 2 * 3` we need to parse it into a tree structure that honors priorities, i.e. that `*` binds stronger than `+`:

```
└──BinaryExpression
    ├──NumberExpression
    │   └──NumberToken 1
    ├──PlusToken
    └──BinaryExpression
        ├──NumberExpression
        │   └──NumberToken 2
        ├──StarToken
        └──NumberExpression
            └──NumberToken 3
```

A naive parser might yield something like this:

```
└──BinaryExpression
    ├──BinaryExpression
    │   ├──NumberExpression
    │   │   └──NumberToken 1
    │   ├──PlusToken
    │   └──NumberExpression
    │       └──NumberToken 2
    ├──StarToken
    └──NumberExpression
        └──NumberToken 3
```

The problem with having incorrect trees is that you interpret results incorrectly. For instance, when walking the first tree one would compute the (correct) result `7` while the latter one would compute `9`.

Generalized precedence parsing

In the [first episode](episode-01.md), we've written our recursive descent parser in such a way that it parses additive and multiplicative expressions correctly. We did this by parsing `+` and `-` in one method (`ParseTerm`) and the `*` and `/` operators in another method `ParseFactor`. However, this doesn't scale very well if you have a dozen operators. In this episode, we've replaced this with [unified method][precedence-parsing].

Compilation API

We've added a type called `Compilation` that holds onto the entire state of the program. It will eventually expose declared symbols as well and house all compiler operations, such as emitting code. For now, it only exposes an `Evaluate` API that will interpret the expression:

C#
```
var syntaxTree = SyntaxTree.Parse(line);
var compilation = new Compilation(syntaxTree);
var result = compilation.Evaluate();
Console.WriteLine(result.Value);
```

Assignments as expressions

One controversial aspect of the C language family is that assignments are usually treated as expressions, rather than isolated top-level statements. This allows writing code like this:

C#
```
a = b = 5
```

It is tempting to think about assignments as binary operators but they will have to parse very differently. For instance, consider the parse tree for the expression `a + b + 5`:

```
    +
   / \
  +   5
 / \
a   b
```

This tree shape isn't desired for assignments. Rather, you'd want:

```
  =
 / \
a   =
   / \
  b   5
```

which means that first `b` is assigned the value `5` and then `a` is assigned the value `5`. In other words, the `=` is *right associative*.

Furthermore, one needs to decide what the left-hand side of the assignment expression can be. It usually is just a variable name, but it could also be a qualified name or an array index. Thus, most compilers will simply represent it as an expression. However, not all expressions can be assigned to, for example, the literal `5` couldn't. The ones that can be assigned to, are often referred to as *L-values* because they can be on the left-hand side of an assignment. In our case, we currently only allow variable names, so we just represent it as [single token] [token], rather than as a general expression. This also makes parsing them very easy as [can just peek ahead] [peek].

Lexical Analyzer

Having a test that lexes all tokens is somewhat simple. In order to avoid repetition, I've used xUnit theories, which allows me to parameterize the unit test. You can see how this looks like in [LexerTests][Lexer_Lexes_Token]:
```C#
[Theory]
[MemberData(nameof(GetTokensData))]
public void Lexer_Lexes_Token(SyntaxKind kind, string text)
{
   var tokens = SyntaxTree.ParseTokens(text);
   var token = Assert.Single(tokens);
   Assert.Equal(kind, token. Kind);
   Assert.Equal(text, token. Text);
}
public static IEnumerable<object[]> GetTokensData()
{
   for each (var t in GetTokens())
      yield return new object[] { t.kind, t.text };
}
private static IEnumerable<(SyntaxKind kind, string text)> GetTokens()
{
   return new[]
   {
      (SyntaxKind.PlusToken, "+"),
      (SyntaxKind.MinusToken, "-"),
      (SyntaxKind.StarToken, "*"),
      (SyntaxKind.SlashToken, "/"),
      (SyntaxKind.BangToken, "!"),
      (SyntaxKind.EqualsToken, "="),
      (SyntaxKind.AmpersandAmpersandToken, "&&"),
      (SyntaxKind.PipePipeToken, "||"),
      (SyntaxKind.EqualsEqualsToken, "=="),
      (SyntaxKind.BangEqualsToken, "!="),
      (SyntaxKind.OpenParenthesisToken, "("),
      (SyntaxKind.CloseParenthesisToken, ")"),
      (SyntaxKind.FalseKeyword, "false"),
      (SyntaxKind.TrueKeyword, "true"),
      (SyntaxKind.NumberToken, "1"),
      (SyntaxKind.NumberToken, "123"),
      (SyntaxKind.IdentifierToken, "a"),
      (SyntaxKind.IdentifierToken, "abc"),
   };
}
```

However, the issue is that the lexer makes a bunch of decisions based on the [next character][Lexer_Peek]. Thus, we generally want to make sure that it can handle virtually arbitrary combinations of characters after the token we

actually want to lex. One way to do this is generate pairs of tokens and [verify
that they lex][Lexer_Lexes_TokenPairs]:
```C#
[Theory]
[MemberData(nameof(GetTokenPairsData))]
public void Lexer_Lexes_TokenPairs(SyntaxKind t1Kind, string t1Text,
                                  SyntaxKind t2Kind, string t2Text)
{
    var text = t1Text + t2Text;
    var tokens = SyntaxTree.ParseTokens(text).ToArray();
    Assert.Equal(2, tokens.Length);
    Assert.Equal(tokens[0].Kind, t1Kind);
    Assert.Equal(tokens[0].Text, t1Text);
    Assert.Equal(tokens[1].Kind, t2Kind);
    Assert.Equal(tokens[1].Text, t2Text);
}
```
The tricky thing there is that certain tokens cannot actually appear directly
after each other. For example, you cannot parse two identifiers as they would
generally parse as one. Similarly, certain operators will be combined when they
appear next to each other (e.g. `!` and `=`). Thus, we only [generate pairs]
where the combination doesn't require a separator.
```C#
private static IEnumerable<(SyntaxKind t1Kind, string t1Text, SyntaxKind t2Kind, string
t2Text)> GetTokenPairs()
{
    foreach (var t1 in GetTokens())
    {
        foreach (var t2 in GetTokens())
        {
            if (!RequiresSeparator(t1.kind, t2.kind))
                yield return (t1.kind, t1.text, t2.kind, t2.text);
        }
    }
}
```
[Checking whether combinations require separators][RequiresSeparator] is pretty
straight forward too:
```C#
private static bool RequiresSeparator(SyntaxKind t1Kind, SyntaxKind t2Kind)
{
    var t1IsKeyword = t1Kind.ToString().EndsWith("Keyword");
    var t2IsKeyword = t2Kind.ToString().EndsWith("Keyword");
    if (t1Kind == SyntaxKind.IdentifierToken && t2Kind == SyntaxKind.IdentifierToken)
        return true;
    if (t1IsKeyword && t2IsKeyword)
        return true;
    if (t1IsKeyword && t2Kind == SyntaxKind.IdentifierToken)
```

```
        return true;
    if (t1Kind == SyntaxKind.IdentifierToken && t2IsKeyword)
        return true;
    if (t1Kind == SyntaxKind.NumberToken && t2Kind == SyntaxKind.NumberToken)
        return true;
    if (t1Kind == SyntaxKind.BangToken && t2Kind == SyntaxKind.EqualsToken)
        return true;
    if (t1Kind == SyntaxKind.BangToken && t2Kind == SyntaxKind.EqualsEqualsToken)
        return true;
    if (t1Kind == SyntaxKind.EqualsToken && t2Kind == SyntaxKind.EqualsToken)
        return true;
    if (t1Kind == SyntaxKind.EqualsToken && t2Kind == SyntaxKind.EqualsEqualsToken)
        return true;
    return false;
}
```

String literals
We now [support strings][ReadString] like so:
```
let hello = "Hello"
```
Strings need to be terminated on the same line (in other words, we don't support line breaks in them). We also don't support any escape sequences yet (such `\n` or `\t`). We do, however, support quotes which are escaped by [doubling them]:
```
let message = "Hello, ""World""!"
```
Type symbols
In the past, we've used .NET's `System.Type` class to represent type information in the binder. This is inconvenient because most languages have their own notion of types, so we replaced this with a new [`TypeSymbol`] class. To make symbols first class, we also added an abstract [`Symbol`] class and a [`SymbolKind`].
Cascading errors
Expressions are generally bound inside-out, for example, in order to bind a binary expression one first binds the left-hand side and right-hand side in order to know their types so that the operator can be resolved. This can lead to cascading errors, like in this case:
```
(10 * false) - 10
```
There is no `*` operator defined for `int` and `bool`, so the left-hand side cannot be bound. This makes it impossible to bind the `-` operator as well. In general, you don't want to drown the developer in error messages so a good compiler will try to avoid generating cascading errors. For example, you don't want to generate two errors here but only one -- namely that the `*` cannot be bound because that's the root cause. Maybe you didn't mean to type `false` but `faults` which would resolve to a variable of type `int`,

in which case the `-` operator could be bound.
In the past, we've either returned the left-hand side when a binary expression cannot be bound or fabricated a fake literal expression with a value of `0`, both of which can lead to cascading errors. To fix this, we've introduced an [*error type*][ErrorType] to indicate the absence of type information. We also added a [`BoundErrorExpression`] that is returned whenever we cannot resolve an expression.

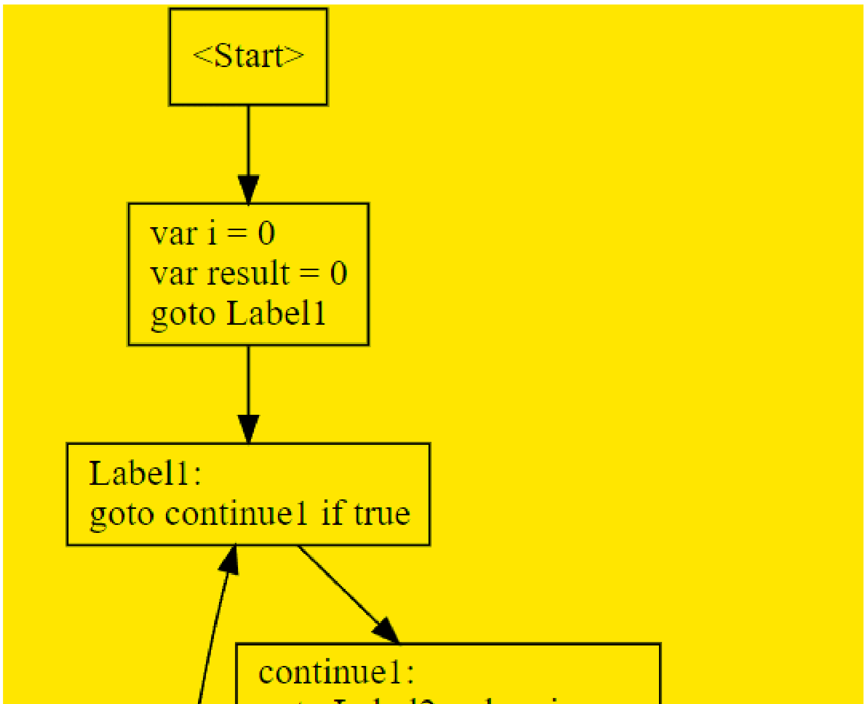
Procedures vs. functions

Some languages have different concepts for functions that return values and functions that don't return values. The latter are often called *procedures*. In C-style languages both are called functions except that procedures have a special return type called `void`, that is to say, they don't return anything.
In Minsk I'm doing the same thing except that you'll be able to omit the return type specification, so the code doesn't have to say `void`. In fact, the type `void` cannot be uttered in code at all:

```
» function hi(name: string): void
· {
· }
(1, 28): Type 'void' doesn't exist.
    function hi(name: string): void
```

Forward declarations

Inside of function bodies code is logically executing from top to bottom, or more precisely from higher nodes in the tree to lower nodes in the tree. In that context, symbols must appear before use because code depends on side effects.However, outside of functions, there isn't necessarily a well-defined order. Some languages, such as C or C++, are designed to compile top to bottom in a [single pass][single-pass], which means developers cannot call functions or refer to global variables unless they already appeared in the file. In order to solve problems where two functions need to [refer to each other][mutual recursion], they allow [forward declarations], where you basically only write the signature and omit the body, which is basically promising that you'll provide the definition later.



Other languages, such as C#, don't do that. Instead, the compiler is using [multiple phases][multi-pass]. For example, first, everything is parsed, then all types are being declared, then all members are being declared, and then all

method bodies are bound. This can be implemented relatively efficiently and
frees the developer from having to write any forward declarations or header
files.
In Minsk, we're using multiple passes so that global variables and functions can
appear in any order. We're doing this by first [declaring all functions] before
[binding function bodies].

To parse or not to parse
In the case of functions without a return type (i.e. procedures), the `return`
keyword can be used to exit it. In the case of functions, the `return` keyword
must be followed with an expression. So syntactically both of these forms
are valid:
```
return
return 1 * 2
```
This begs the question if after seeing a `return` keyword an expression needs to
be parsed.
In a language that has a token that terminates statements (such as the semicolon
in C-based languages), it's pretty straightforward: after seeing the `return`
keyword, an expression is parsed unless the next token is a semicolon. That's
what [C# does too][roslyn-return].
But our language doesn't have semicolons. So what can we do? You might think we
could make the parser smarter by trying to parse an expression, but this would
still be ill-defined. For example, what should happen in this case:
```
return
someFunc()
```
Is `someFunc()` supposed to be the return expression?
I decided to go down the (arguably problematic) path JavaScript took: if the
next token is on the same line, we [parse an expression][parse-return].
Otherwise, we don't.

About IL
Technologies like the .NET runtime are often called *virtual machines* because
their instruction set (also called byte code) isn't targeting a physical CPU but
an imaginary one. It's the responsibility of the just-in-time (JIT) compiler to
produce the machine code. In some cases, this is also done ahead-of-time (AOT),
for example, by using a ready-to-run or an AOT compiler. Having this separation is
common in compiler pipelines because it allows to support of multiple languages
(e.g. C#, VB, F#) and multiple CPU architectures (e.g. x86, x64, ARM32, ARM64)
without having to build all combinations. Adding a new CPU target only requires
adding a single compiler from IL to machine code as opposed to having to build
this for each language. It also simplifies letting different source languages
reference each other.
The .NET byte code is called [common intermediate language (CIL)][CIL], but most
people just call it IL.

Encoding of IL instructions

IL instructions are ultimately bytes. Most instruction sets try to optimize for size while also making sure that the format is extensible and flexible. IL is no different. Instructions come in two forms, by themselves or with a parameter, which is called an intermediate. The intermediate is used as an additional parameter for the instruction, for example, the local variable being stored, the method being called or the literal being loaded. IL only allows for a single intermediate, which is why many instructions don't use arguments directly but instead use the evaluation stack. For example, the `add` instruction doesn't take two arguments but instead takes them from the stack.

The general instruction for loading 32-bit integer values is `ldc.i4`. The intermediate is the value, for example, `ldc.i4 42` loads the value `42`. Since some values are extremely common in programs (such as `0` and `1`) there are special instructions that don't need an intermediate because the instruction itself represents the value being loaded, for example, `ldc.i4.0` just loads the value `0`. This reduces the size of IL.

As a compiler writer, dealing with these special encodings can be tedious but fortunately, we don't have to. We're using `Mono. Cecil` for emitting IL and it has the handy `body.OptimizeMacros()` method which will replace instructions accordingly.

Backpatching

All control flow is expressed as jumps. In assembly language, the target of a jump is usually a label. However, labels are only used to make things readable for a human. The underlying instruction sets don't allow for labels; instead, addresses are being used. IL is no different: the jump address is a byte position within the IL stream.

This poses a problem when the target of a jump is to a label that exits further down, for example

```
    gotoTrue a <= b break
    ...
break:
    return 4
```

At the time we're asked to emit the instruction for `gotoTrue a <= b break` we don't know the address of `return 4` yet (because it depends on the size of all instructions in between).

The usual approach for this is simple: we emit the instruction with a bogus address and patch it later when we know the address for `return 4`. This is possible because the size of the jump instruction doesn't depend on the actual value of the address (for example, because it's expected to be a 32-bit number).

Since we're using `Mono. Cecil` we don't have to deal with addresses directly. But the problem is similar. The jump opcode takes an argument of type `Instruction` which we haven't constructed yet. So we're just manufacturing a fake instruction (a nop, which is an instruction that does nothing). Then [we're recording the instruction in a fix-up

Constant folding

It's pretty common in source code to have expressions that involve constants. However, right now we're evaluating their value at runtime. We can improve this by detecting such expressions during compilation and pre-computing their values. This technique is called [constant folding]. For example:

```JS
let x = 4 * 3
```

We know that `x` has the value `12`. This gets more interesting when we combine this with boolean expressions, such as:

```JS
let x = 4 * 3
while x > 12
{
  print(x)
  x = x - 1
}
```

Since we understand that `x` has the value `12` (and `x` cannot be changed), we also know that the `while` loop will never execute. We could use that information to provide a warning ([#136]) because there is a decent chance that the developer made a mistake.
But this allows us to remove the entire body of `while` and, in this case, also the condition as well. You might wonder what I mean by "in this case". Well, expressions can have side effects. For example:

```JS
while process(x) && x > 12
{
  doSomething(x)
}
```

Even though we know that the value of the condition is always `false`, not evaluating it might change the observable behavior of the program. For example, the way it is written, the `process()` function would be called once before the loop condition is evaluated to `false`. Depending on language semantics removing this call completely might not be desirable.
In Minsk, I'm declaring this as undesired behavior. However, we're currently not handling this correctly ([#125]). We'll fix this when we also handle short-circuit evaluation ([#111]).

Line-independent syntax highlighting

Right now, we don't have any tokens that span multiple lines. This makes syntax highlighting rather simple: we can tokenize each line independently. With the advent of multi-line comments, this is no longer the case. Consider I have several lines of code. Now let's say I insert a new line at the beginning and

start typing `/*`. We now have to repaint multiple lines because they are all considered to be part of the comment.
A common trick that fast syntax highlighters use is that they will track a state per line. Usually, that state is just an integer representing the initial state the tokenizer should be considered in when tokenizing the next line. In our case, there would be only two: regular state and an in-comment state.
We're doing a simpler version of that and simply say that [the state] is the same for all lines: the fully tokenized input.
```C#
private sealed class RenderState
{
  public RenderState(SourceText text, ImmutableArray<SyntaxToken> tokens)
  {
      Text = text;
      Tokens = tokens;
  }
  public SourceText Text { get; }
  public ImmutableArray<SyntaxToken> Tokens { get; }
}
```
When rendering, [we pass the array of lines, the index of the current line, and the previous line's state][RenderLine]. If the previous line's render state is `null`, we tokenize the input:
```C#
protected override object RenderLine(IReadOnlyList<string> lines, int lineIndex, object state)
{
  RenderState renderState;
  if (state == null)
  {
      var text = string.Join(Environment.NewLine, lines);
      var sourceText = SourceText.From(text);
      var tokens = SyntaxTree.ParseTokens(sourceText);
      renderState = new RenderState(sourceText, tokens);
  }
  else
  {
      renderState = (RenderState) state;
  }
  // ...
}
```
For the actual rendering we only look at tokens that [overlap with the span of the current line][tokenLoop]. We also have to make sure that we trim the token to the line start and line end:
```C#
var lineSpan = renderState.Text.Lines[lineIndex].Span;
for each (var token in renderState.Tokens)
```

```
{
    if (!lineSpan.OverlapsWith(token. Span))
        continue;
    var tokenStart = Math.Max(token.Span.Start, lineSpan.Start);
    var tokenEnd = Math.Min(token.Span.End, lineSpan.End);
    var tokenSpan = TextSpan.FromBounds(tokenStart, tokenEnd);
    var tokenText = renderState.Text.ToString(tokenSpan);
    // Print token
}
```

Leading vs. trailing trivia

Earlier I wrote that a token can have leading and trailing trivia. You might wonder why that is. We could get away with only having leading trivia. All trivia that follows the last token in a file will be leading trivia for the end-of-file token.

Having both leading & trailing trivia allows us to associate them with the token that makes more sense for the developer:

* If a comment is on a line by itself, it usually refers to the following code.
* If a comment is on a line with code, it usually refers to the code before it.

Let's look at a few examples:

```JS
// Comment 1
function Foo(a: int, // Comment 2
             b: int) // Comment 3
{
    let x = a /* Comment 4 */ + /* Comment 5 */ b
    /* Comment 6 */ let y = x // Comment 7
    // Comment 8
}
```

According to the rules stated above the comments are associated with tokens as follows:

Comment	Leading/Trailing	Token
Comment 1	Leading	`function`
Comment 2	Trailing	`,`
Comment 3	Trailing	`)`
Comment 4	Trailing	`a`
Comment 5	Trailing	`+`
Comment 6	Leading	`let`
Comment 7	Trailing	`x`
Comment 8	Leading	`}`

Adding a parent property to an immutable tree

For an IDE it's super useful to be able to walk the syntax nodes from the leaves upwards. This allows us to build a single API that, given a position, can return the token it is contained in. Calling code can then walk the parents until it finds the node it's interested in (for example, the containing expression, statement, or function declaration).

However, to construct an immutable tree, one has to construct the children before the parent, which is exactly what our parser does. This raises the question of how one can have a `Parent` property on `SyntaxNode`.

The answer is: that we can cheat.

The parser is executed from the constructor of `SyntaxTree`. This allows the parser to pass in the `SyntaxTree` to each node. Since the `SyntaxTree` provides access to the syntax root, we can add a method on `SyntaxTree` that constructs a dictionary from child-to-parent, which we'll produce upon first request, also known as [lazy initialization].

Lazy initialization is a common technique for immutable data structures but one thing to watch out for are race conditions. One of the reasons why we make things immutable is so that we can freely pass them around to background threads and not worry about multi-threading bugs because the data structures can't be modified.

Strictly speaking, lazy initialization violates this because we have a side effect that writes to an underlying field. The trick is to make sure this side effect is unobservable. We achieve this by using `Interlocked.CompareExchange` from [SyntaxNode.GetParent][syntaxtree-getparent]:

```C#
internal SyntaxNode? GetParent(SyntaxNode syntaxNode)
{
    if (_parents == null)
    {
        var parents = CreateParentsDictionary(Root);
        Interlocked.CompareExchange(ref _parents, parents, null);
    }
    return _parents[syntaxNode];
}
```

Logically, this code

```C#
Interlocked.CompareExchange(ref _parents, parents, null);
```

is equivalent to this code:

```C#
if (_parents == null)
    _parents = parents;
```

but it does so in an atomic fashion, that is to say, between the check and the assignment no other thread will have the opportunity to write to the underlying field.

The net effect is this: some thread will be the first to assign to this field

and all other threads will see this value. However, please note that multiple
threads might have been called `CreateParentsDictionary` but only one thread
will succeed in storing the result in the `_parent` field. So this only works if
multiple threads can call `CreateParentsDictionary` without problems, which
generally means it shouldn't have any observable side effects either.
In the literature, `CompareExchange` is often referred to as [compare-and-swap].
You might wonder why it matters that only one thread succeeds here. In the end,
it doesn't matter if we were to overwrite the previous dictionary because it
contains the same information. That is true here. But sometimes you construct
objects that are publicly visible. Changing already observed instances to new
ones (even if they are logically equivalent) can break observers because object
identity changes. So don't do that.

CONCLUSION –

Basically, in this I developed MY COMPILER by using C++. This report contains Minsk, a handwritten compiler in C#. It illustrates basic concepts of compiler construction and how one can tool the language inside of an IDE by exposing APIs for parsing and type checking.