# Challenge Mode: Compilers – Day 0

Intro, expectations, material overview, grammar

## Syllabus

#### Description

In this course, you will build a complete compiler and virtual machine for a scripting language. All of this will be built in standard C++ using no language construction tools such a Lex or Yacc, rather you will build the components from scratch using just the core language and the STL. The reason we take this approach is to reinforce how your day to day tools work to the point that they feel natural.

#### Beta

This course is in beta which means all of this is a best guess on approach. Feedback is necessary to improve the experience, please be generous with it. Additionally, all timelines are estimates since only data will allow me to refine them.

#### Meeting

The course will have a weekly meeting until complete. It takes about 10 weeks in practice, though if we have critical mass for conflicts then we will bump some weeks extending the end time further out. Each session will be 3 hours in length with the first half being the lecture and the second portion being a lab session. Outside of class you are expected to put in up to 10 hours of time a week and this will be the baseline that expected progress rate is measured against.

For discussing things out of class, I have setup a Discord server where all documents and discussions should take place. This will help reduce repeat questions, help people learn from each other, and provide continuity across course instances.

#### Participation and Attendance

Presence and completion of course work is required for continued participation since offering the course is not without commitment on my part. It is understood that we all have real jobs and real lives and we will punt a class instance when necessary. I can also work with people if they need to miss an instance due to “life”.

#### Course Progression

The rough progression of the course is:

* Research and customize language
* Build lexical analyzer
* Build parser
* Build bytecode generator
* Build virtual machine
* Refine language
* Final project

#### Completion

You will be considered to have completed the course when you have successfully built your compiler stack and virtual machine, and can demonstrate final project functionality.

## Why? Why do this?

#### For you

* True learning comes from projects, not presentations
* Very few programmers that I’ve met have ever written large scale, complex software on their own
* Compiler courses at the university level focus heavily on language theory, not practice, which limits utility
* Becoming familiar with the inner workings of our tools can impart technical intuition that can greatly assist in problem solving in our field

#### For me

* Meeting more of the Xbox team, learning who they are and how they think
* I love seeing how other people solve problems and the courses I’m designing *requires* people to solve things that are new to them

#### For the future

* Teaching is how we can mold the future
* Building a culture of learning is a self-fulfilling prophecy. Some of the greatest things you can learn are also some of the most impenetrable without a guide: compilers, game engines, etc.
* If current University trends continue away from our organizational needs, our only option will be to hire the smartest people we can find and train them ourselves

## How It Works

The course is broken into four phases: language design, compiler, virtual machine, and final project. Each phase will bleed into the next as building the major components is an iterative process.

#### Phase 0: Language Design

This is the squishiest part of the course. Here we will look at existing scripting languages: how they work, their features, and the role they fill. At this point we will lay out the base language requirements for your language and you’ll choose how to achieve them. This phase will finish with you customizing a base grammar to your goals and preferences.

#### Phase 1: Compilation

This is where the real work starts and the largest phase. The compiler has three major components: the lexical analyzer (tokenizer), the parser, and the bytecode generator. We will construct these in that order while developing test suites for each one. This is a very iterative phase and it is all but guaranteed that you’ll be re-factoring heavily as you progress. At the end of this phase you will be taking source code input and generating preliminary bytecode for your virtual machine.

#### Phase 2: Virtual Machine

Here you will build the runtime for your language. It will be a complete stack based virtual machine with simulated CPU, ref counted memory management, built in types, foreign function interface, etc. Here you will revisit the rest of the stack as you find issues with your language and technology design. Once done, you’ll be able to demonstrate running code for all features of your language and will be well positioned to extend your system as necessary.

#### Phase 3: Final Project

The exact specifics of this are not known yet. My current guess is that for this phase, you will add some form of class support to your language and another feature of your choosing from a list of examples such as an error system, FFI auto generation, etc. Completion of this phase will leave you with a complete, custom built scripting language.

## Books

There is only one book I recommend for the course so far however it isn't necessary for participation:

* *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages* by Terence Parr – it provides the clean coverage of parsing without getting bogged down in the many academic topics that surround it

## Your Language

At a high level, your language is going to be an imperative, dynamically typed, roughly C-style language. It will run in a stack based virtual machine with reference counting as the memory management primitive. Its base required initial features are:

* Value type support: number, string, list, map, boolean
* Control flow: if, for, while
* Function declaration and use
* Math operators: + - \* / % ++ -- !
* Boolean operators: && ||
* Assignment operator: =

Other possible features include:

* Additional control flow: switch, do, break, continue, foreach
* Classes (prototype, concrete, etc.)
* Tuples
* Atoms
* Enums
* Functions as first class types
* Pattern matching
* Exceptions
* Etc.

// TODO: Example code

## Review Existing Languages

The first step to customization the grammar to be your own language will be looking at existing languages and scripting languages especially for those of you who haven’t used them.

* Lua
* Ruby
* Python
* Rust
* C#
* Lobster
* Others?

## Language Features

Language features span three broad groups:

* Syntax: these are features around how your language looks and feels, e.g. its control flow structures, type syntax, etc.
* Runtime: features which are inherent to the virtual machine such as supported bytecode, garbage collection, or reflection
* Library: the standard libraries of languages are now a critical feature to the languages that you choose

When thinking about your language, think about which group things you are interested fall into. It helps to keep a list of features you want to support so you don’t close it off by some minor seeming engineering decision.

## Grammar

At a base level, your grammar is a collection of rules (called production rules or just productions) that collectively define what is valid syntax for your language. For our needs we will be using what is called a Context Free Grammar (CFG)[1] where each rule has no attached predicates. CFGs are excellent for defining programming languages since it establishes a limit on complexity on what can be defined by the rule set.

#### Structure

Unsurprisingly there are common methods for defining grammars. For the course we will be using a derivative of the Extended Bachus Naur Form (or EBNF)[2] for defining ours. EBNF is a syntax definition for expressing a grammar’s rules. The general form of a rule is:

*<name> = <replacement>;*

Each of your grammars will have a root rule with a name such as *grammar* or *program*. This rule will reference other rules, and they will reference other rules or themselves recursively, and so on. Other than the root rule, no other rule exists without being referenced by another. This gets back to the name production since each rule is a name that produces the replacement.

The control structures you have for the replacement portion of your rules are:

* Or: |
* Concatenation: ,
* Optional: []
* Zero or more: {}
* Grouping: ()
* Keyword: “<word>”
* Text description: ? blah ?

An example subset for a simple grammar for a language such as ours:

program = { func };

func = “func”, ident, “(“, [ ident, { “,”, ident } ], “)”, block;

letter = ? all visible ansii upper and lowercase letters ?;

digit = “0” | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";

ident = ( letter | “\_” ), { letter | digit | “\_” };

block = “{“, { statement }, “}”;

Ultimately your grammar is going to be translated into code to implement your parser. As you work on the grammar you should think through applying the rules to your language and ensure that it can produce a tree format without ambiguity in the rule application. This will take time and mental gymnastics but time spent here is more time saved later.

For the style of parser we will be implementing a pitfall that can (and will) happen in your grammar is left recursion. This a rules such as in the following where the rule self-references itself on the left side of a sub expression in its production:

number = digit, { digit };

math2 = math2, ( “\*” | “/” ), math2 | number;

math1 = math1, ( “+” | “-“ ), math1 | math2;

While this type of rule isn’t wrong in general, it isn’t supported with our parser and so must be worked around. This is easily fixed by changing it to:

number = digit, { digit };

math2 = number, ( “\*” | “/” ), math2 | number;

math1 = math2, ( “+” | “-“ ), math1 | math2;

This simple change removes the left recursion, supports the order of operations for our parsing structure, and is an example of expression parsing.

#### Goals

Your objective when creating the grammar for your language is to create your rules in such a way that you narrow the valid input syntax without reducing the expressiveness of your language. Finding the right balance here will take time and likely won’t be complete until your parser component is done. An example where expressiveness of the language trumps restrictiveness in the grammar commonly occurs when defining the expression rules. It is important to remember that many syntax errors are caught during different stages of compilation and some of your syntax errors won’t be found until bytecode generation or even runtime.

Beyond this, requirements for the first pass on your grammar are:

* All operators and control structures you wish to support must be present
* Order of operations must be accounted for (C style)
* Cannot have left recursion
* Expressions are statements, statements are not expressions

I recommend ignoring the left recursion goal with the initial implementation of your grammar and focus just on the syntax you need to support in terms of constants, operators, control structures, declaring functions, making function calls, and basic types such as number, booleans, strings, etc. Make sure that it supports application against some complete example code.

## Homework

Read up on existing programming languages, what their designers think is great about them, and their design rational. Think about syntax, built in features, libraries, runtime functionality, etc. Document everything you plan to support for review in the next course. If there is any short-term syntax you’d like to support, incorporate it into your grammar before the next class.

### Base Grammar

all\_characters = ? all visible ansii characters ?;

letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i"

| "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r"

| "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "A"

| "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J"

| "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S"

| "T" | "U" | "V" | "W" | "X" | "Y" | "Z";

terminal = '"', { all characters }, '"';

digit\_excluding\_zero = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";

digit = "0" | digit\_excluding\_zero;

natural\_number = digit\_excluding\_zero, { digit };

number = "0" | [ "-" ], natural\_number, [ ".", digit, { digit } ] | [ "-" ], "0", ".", digit, { digit };

bool = "true" | "false";

constant = terminal | number | bool | list | map;

ident = ( letter | "\_" ), { letter | digit | "\_" };

key\_value\_pair = terminal, ":", expression;

map = "{", [ key\_value\_pair, { ",", key\_value\_pair }, [ "," ] ], "}";

list = "[", [ expression, { ",", expression }, [ "," ] ], "]";

index = "[", expression, "]";

call = "(", [ expression, { ",", expression } ], ")";

block = "{", { statement }, "}";

while\_loop = "while", "(", expression, ")", statement;

for\_loop = "for", "(", expression, ";", expression, ";", expression, ")", statement;

if = "if", "(", expression, ")", statement, { "elif", "(", expression, ")", statement }, [ "else", statement ];

return = "return", [ expression ], ";";

parens = "(", expression, ")";

ex\_term = ident | constant | parens;

ex\_postfix = ex\_term, { ( index | call | "++" | "--" ) } | ex\_term;

ex\_prefix = ( "!" | "++" | "--" | "-" ), ex\_prefix | ex7;

ex\_mul = ex\_prefix, ( "\*" | "/" | "%" ), ex\_mul | ex\_prefix;

ex\_add = ex\_mul, ( "+" | "-" ), ex\_add | ex\_mul;

ex\_compare = ex\_add, ( "<" | ">" | "<=" | ">=" ), ex\_compare | ex\_add;

ex\_concat = ex\_compare, "#", ex\_concat | ex\_compare;

ex\_equals = ex\_concat, ( "==" | "!=" ), ex\_equals | ex\_concat;

ex\_and = ex\_equals, "&&", ex\_and | ex\_equals;

ex\_or = ex\_and, "||", ex\_or | ex\_and;

expression = ex\_or, ( "=" | "\*=" | "/=" | "+=" | "-=" | "%=" | "#=" ), expression | ex\_or;

statement = [ expression ], ";" | block | while\_loop | for\_loop | if | return;

func = "func", ident, "(", [ ident, { ",", ident } ], ")", block;

program = { func };