SPENGERGASSE

Language Design

Ariel Simulevski

Version 1.0, 2018-12-31

Table of Contents

Abstract

This paper aims to serve as an introduction into language design/development. Both the design aspect as well as the technical aspect are explained.

In the end of this text, all of the learned skills will be utilized by creating a small, custom programming language.

It is hoped this paper will inform others about language development, language design and the rules to the latter.

Introduction

The list of general purpose programming languages has been growing recently. With it, a staggering amount of immature and, generally, out of place languages. Of course, there are some really good new languages (Kotlin, Go, Rust), but others (such as Solidity, JavaScript or Python) have features or design flaws that make them seem out of place or unliked by some.

Language design and intelligent language features are now more important than they’ve ever been. With the amount of languages out there, creating a new language with features that sets it apart and doesn’t make it seem "ugly" or "irrelevant" is quite a challenge. This paper will aim to introduce you to language- and language-feature design.

What is a programming language?

What many people don't understand is, that a programming language is nothing more than an agreement. An agreement between you, the programmer, and the creator of a given language, that states how a program in the given language has to be written (how the instructions are called, how to write functions, loops, declare variables etc.) so that the runtime understands it.

Source code itself is also nothing special. It is just bytes in a, in most cases, UTF-8 encoded text file, which are arranged in a manner so that they form commands which the runtime of a given language can understand.

Nearly everything can be considered a language. If we define a language so that

- the character `*` followed by an identifier, creates a variable
- the character `+` followed by an identifier, adds one to a variable
- the characters `:=` preceeded by an identifier and followed by a numeric value or an identifier, assigned a value to a variable
- the characters `+=` preceeded by an identifier and followed by a numeric value or an identifier, added a value to a variable
- the character `=` preceeded by an identifier and followed by an identifier, compares two variables
- an identifier followed by the character `:`, creates a label

  and

- the character `<` followed by a lable name, jumps to a label if the last comparison evaluated to `1`

this:

```
*a
a:=0

*b
b:=0

loop:
b=10
< next

a+=b
+b

1=1
< loop
next:
```

would be valid source code (in fact, the language could even be considered Turing complete). We won't be able to execute it, of course, as we don't have a runtime for that language, but it is a defined language, nonetheless.

Compiled vs Transpiled vs Interpreted

For a program written in a language to be executed, you need a runtime. A runtime is the basic execution model of any implementation of any language.

When we speak of runtime levels, we refer to code that runs closer, or further away (in means of abstraction) from the hardware. This, in turn, means that a system has to have a concept of "closer or further away from hardware".

Compiled

A compiler scans the entire source code and translates it into lower level code. That doesn't always mean converting the source code into machine code, but rather translating the code into a very primitive version of itself. Sometimes, code is compiled into bytecode (primitive code than can be executed in a virtual machine).

Examples

- **GCC** [1]: C++ → Machine code
- **Roslyn** [2]: C# → CIL (Common Intermediate Language, C# bytecode)
- **javac** [3]: Java → Java Bytecode

Transpiled

A transpiler also scans the entire source code, but instead of translating it into lower level code, the transpiler translates it into another language on the same execution level. This language can than be compiled, transpiled or interpreted itself.

Examples

- **Babel** [4]: ES6 → JavaScript
- **CoffeeScript** [5]: CoffeeScript → JavaScript
- **emscripten** [6]: LLVM → JavaScript

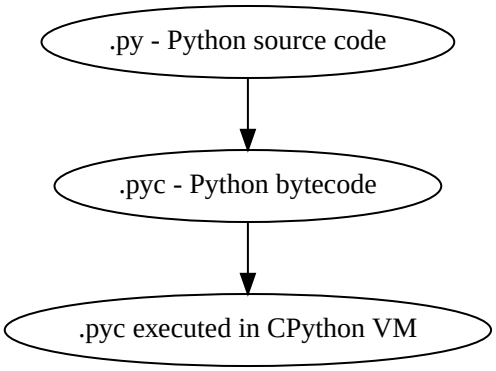
Interpreted

An interpreter is practically not comparable to a compiler or a transpiler. Instead of translating the language, an interpreter goes through the source code, line-by-line, and tries to figure out what the program is supposed to do as it goes.

In some cases, the interpreter processes bytecode instead of source code. An interpreter that interprets bytecode is called a virtual machine (or VM for short).

When an interpreter processes bytecode, it first needs to compile whatever language it interprets into the bytecode (which means that such an interpreter always has a compiler in its execution pipeline).

An example for that process would be CPython [9].



Python source code to execution lifecycle

In recent years, people started to implement JIT (or Just-in-Time) compilers into their interpreters.

These JIT compilers can figure out what parts of a program are resource-intensive and then compile these parts of the program (at runtime, meaning while a program is being executed), from bytecode into machine code. Everything else is still being executed in a VM. Whenever the VM needs to call code that has been "JITed", it calls those, now native, functions.

Examples

- **V8** [7]: JavaScript
- **HHVM** [8]: Hack & PHP

Definition vs Implementation

Before we start, we first need to understand the difference between language definition and language implementation. These two are often confused and thus, people say things like "C++ is a fast language" or "Python is really slow".

But languages themselves, have nothing to do with speed. Language implementations do. Of course, certain languages tend to encourage certain

runtime systems, while language constructs of others might make it harder to have a faster runtime. But, in theory, every language can be as fast as any other.

> The reference implementation is the implementation made by the creator of a language. It, usually, gets new language features first and is always compliant with the standard (because it is the standard).

One could make an interpreted version of C++, which would be slower than the reference implementation (GCC, compiled). Same for Python. It would theoretically be possible to write a Python compiler which would have significantly faster runtime performance than its reference implementation (CPython, interpreted).

In fact, there are many non-reference implementations of languages out there (like the Just-in-Time compiled Python implementation, PyPy [10], or the C++ interpreter Cling [11])

What is Turing completeness?

For us to understand Turing completeness, we need to go back in time. To the 1940s to be precise. Back then, the second world war was at its height, the Nazis seemed undefeatable. No one could predict their strategy. The only way to do it was cracking their encryption algorithm, the "ENIGMA". The challenge seemed so completely out of scope that for a while, British intelligence tried to just guess what the code might be. Until Alan Turing came along and created a computational device that was able to figure out the encryption keys of the "ENIGMA".

Fascinated by the idea of a universal computing device, Turing first turned his attention to designing a general purpose computing machine in 1936. This was when Turing first formulated the idea of the "Universal Computing Machine" [12] (now simply known as a "Turing machine"). A mathematical model that defines an abstract computation device. The machine works by modifying symbols on a strip of tape according to opcodes (a table of rules). Even though the model was simple, it could, in theory, simulate any given mathematical algorithm.

Turing wasn’t the only one who formulated such an idea. In 1931, Kurt Gödel had already published a similar mathematical formalism in his paper on the "Gödel’s incompleteness theorems" [13].

A Turing complete language describes a language that has, at least, the same computational capabilities as Alan Turing had with his "Turing machine". To put it simply:

A Turing complete language can emulate a universal Turing machine.

That also means that two computational devices P and Q are equal if P can simulate Q and Q can simulate P. This is called Turing equivalence.

The EBNF (Extended Backus-Naur-Form)

Invented by Niklaus Emil Wirth, EBNF (named after John W. Backus and Peter Naur) was meant as a way to describe the grammar of any given language. It is based on the simpler, less advanced Backus-Naur-Form (or BNF).

EBNF describes the grammar of a language. It is the language of languages, so to say. In EBNF, there are two main unit types:

- Tokens (sometimes called lexemes)

  and

- Statements

Operators

- `|` ..... "or" operator. Either left side or right side.
- `,` ..... Concatenation. Concatenates two values.
- `[ ⋯ ]` ... Optional block. Values inside this block are optional.
- `{ ⋯ }` ... Repetition block. Values inside this block can repeat.
- `" ⋯ "` ... String value.
- `;` ..... Terminates an operation

Tokens

Tokens are fixed values. They're the smallest unit in an EBNF definition.

```
zero = "0" ;
digit_without_zero = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

Statements

Statements are a bundle of multiple tokens. Every statement can be broken up into sub-statements and/or tokens.

```
hex_prefix = zero, "x" ;
digit = zero | digit_without_zero ;
```

Every EBNF definition also has a start point. This starting point is often called "program", "main" or "statement".

If we want to define a program that accepts decimal or hexadecimal numbers that don't have leading zeros as input, using our previously defined tokens and statements, our EBNF could look like this:

```
program = number | hex_number ;
number = ( digit_without_zero, { digit } ) | zero;
hex_digit_without_zero = digit_without_zero | "a" | "b" | "c" | "d" | "e" | "f" ;
hex_digit = hex_digit_without_zero | zero;
hex_number =  hex_prefix, ( hex_digit_without_zero, { hex_digit } ) | zero;
```

For this definition, values like `0x1f`, `0x0`, `20`, `1` and `0` would be valid, while `01`, `0xr` or `x12` are invalid.

Different types of programming languages

Before we get into actually creating a programming language, we need to know what kind of language we want. We can generally differentiate three different kinds of programming languages:

A programming paradigm is a style of programming. It is a way of thinking about a solution. Programming paradigms do not refer to a specific language, but rather to a type of programming language (to a way to program).

Procedural

According to most people, procedural languages are the most basic form of programming languages. In a procedural language, the programmer describes what the program is supposed to do step-by-step. There are basic procedures, known as subroutines or functions, and more often that not, one will be able to define data structures in a procedural language, but that is as feature-rich as it gets.

Examples

- C
- Fortran
- Bash

Object oriented

Object oriented programming, OOP for short, builds on top of procedural programming. The central programming paradigm in OOP is based on the concepts of objects. These objects can contain data or procedures. When a procedure is associated with an object, it is referred to as a method.

Examples

- C++
- Java
- Python

Functional

Functional programming aims to model a program as the evaluation of mathematical functions. Generally, state-changing and mutability of data is avoided. One of the central programming paradigms is currying [14]. Currying, which was invented by Moses Schönfinkel in 1928 and named after Haskell Brooks Curry, a famous mathematician, is about passing data to a function to generate another function as a result. This resulting function can then be used to do other calculations. Since mutability and state-changing is avoided, functions should not have any shared state or side-effects.

Examples

- F#
- Haskell
- Clojure

The rules of good language design

Simplicity

When we talk about simplicity in language design, we mean simple as in that there should be a small number of basic principles. It doesn't matter whether these basic principles themselves are simple. But having too many different paradigms in one language won't make it more powerful. It will make the language harder to use and thus less powerful.

> You might not get all of this year's cool features but it will be easier to understand.
>
> — Robert Virding, On Language Design

Consistency

The basic principle of language consistency is the following: things should always look like what they're doing and things should always look like they fit in. Taking a C-like syntax and putting it into a Python-esque language will look and feel wrong; Merging an object oriented paradigm into a functional language won't work, etc.

Example

C-like syntax in Python-esque language

```
def check(x):
    if x is true {
        print x
    }
```

Syntax

When starting to use a new programming language, many people complain about the syntax of the language not looking like **their** favourite language. But the truth

is that not having the same syntax as another language is not a bad thing. People writing in your language might have to re-learn certain things, but having a unique syntax gives you the flexibility to have unique paradigms and features. Just ripping off another language's syntax (how popular it might be) also means ripping off that language's flaws (which we want to avoid at all cost).

The syntax of a language should reflect its semantics and its paradigms. Taking another languages syntax with different semantics and paradigms **will** lead to problems. But this doesn't mean that you have to completely change everything when designing a language. Certain keywords like `for`, `while` or `class`, names of primitive types like `int` or `string`, or common methods should stay the same.

This, in no way, means going over the top with ones syntax. Proving alternative syntaxes for the same problem is bad. Not only does it make learning the language harder, it also makes a language less opinionated. Same goes for syntax for special cases.

Explicitness

We all love not having to write a lot of code and just letting the runtime guess what we could have meant, but being in-explicit, just for the sake of writing code that has a couple fewer bytes, is no good.

A compiler could, in theory, differentiate between an `=` in an assignment and as an equality comparison operator.

That means, that a language could exist where this:

```
a = 10

if a = 10:
    print("a is 10")
```

is valid code. And with a decent enough recursive descent parser, we'd be able to make something like this valid. Which certainly doesn't mean that we should.

How to write a programming language?

Now that we know the rules to good language design, and know what not to do when developing a programming language, we can start with writing our own little language.

Pre-requirements

Before we start writing your programming language, we need to ask ourselves some questions. This will make the process of conceptualizing features and actually developing our language easier and will give us a clear definition as to what we actually want to achieve.

Technical aspects

What kind of programming language is it?

It is generally agreed upon that procedural languages are the easiest to develop. There are no classes, interfaces or other code abstractions (like polymorphism). Functional constructs, such as currying, are also not supported. Simply put: there are less features one has to support, thus, the language itself is not as complicated.

Object oriented or functional language are both harder to develop and to conceptualize, but are often far more powerful than procedural ones.

Is it compiled or interpreted?

The question, whether your language should be compiled or interpreted regards your reference implementation. While compiling tends to have greatly improved performance over interpreted languages, interpreting a language comes with higher flexibility.

For a reference implementation, interpreting a language makes more sense. Interpreters often have a bigger feature set and are easier to debug than compilers, thus making them better for trying out a language. [1: Personal

opinion]

Non-technical aspects

Why are you writing it?

As mentioned in the introduction, the list of programming languages is nearly endless. There is a programming language for almost every use-case out there. So why write a new one?

There is no right or wrong to this question. The answer "Just for fun" is as correct as "Because I thougt of a new language feature that will revolutionize how we program". Nevertheless, it is of utmost importance that you ask yourself that. That is, because developing a programming language is all about focus. It is about focusing on the thing you want your language to do.

If your language is designed for a certain thing, trying to make it do something completely different will just complicate the issue and unless you're either really lucky or really good, you'll end up with something extremely complex.

Who are you writing it for?

Knowing your user base is always important. Whether you develop an online shop or a programming language doesn't really matter. What does matter is that you know who will be using your language. Writing a language for statistical computing, like R [15], for instance, is something completely different from writing a general purpose programming language.

While R mostly used by students or scientists, a language like C# is mostly used by software engineers. R doesn't need to have the same enterprise capabilities like C# and C# doesn't need to have the same ability to express complicated mathematical formulae, like R does.

Developing a language

Giving the language a name

As irrelevant as this might sound, the name is an important aspect of every programming language. It should be memorable and easy to pronounce. The file extension you choose for source files should not be taken.

Defining the language

When defining a language, one usually starts by writing an EBNF. This isn't always necessary, but is recommended in most cases.

The core parts of the language are defined first. This includes primitive types, variables, functions and loops. Then, method calls, class structures and other high-level constructs are defined.

Define what differentiates your language from other languages, syntax wise. Don't, yet, talk about features.

Defining the feature set of the language

After the basic functionality is clear, you can start defining the feature set of your language. Code samples are always welcome. Explain why your users should utilize said feature and why your feature is a better way of solving a problem.

Lexer

As previously mentioned, an EBNF has tokens and statements. The lexer parses a string and converts the characters of said string into tokens.

Parser

The parser takes in the lexer output as a stream of tokens and converts them into statements. There are many different types of parser, but the most common one is the recursive descent parser. A recursive descent parser operates as a finite

state machine. One of its features is parsing sub-statements while parsing a statement.

Adding a feature to a language

Adding features to a language, after the fact, is more complicated than it appears to be. Adding too many features might make the language cluttered and unusable. Certain features might not look right in the language (from an aesthetic standpoint, that is) and others might just not be technically possible. The most important part is keeping focus. Focus on what you want your language to be and avoid adding features that don't fit in.

Because there is no feature that is not a limititation on something else, be very restrictive about adding features to a language.

Defining a feature

To add a feature, we first need to define it. Say we want to add pipelines to the C# programming language.

We can't implement this feature by utilizing existing functionalities as there are no macros in C#, so we need to define a new operator. Let's take the well known `|>` operator from F#. Now that we know what we're going to add into our language, we need to write a basic EBNF for that.

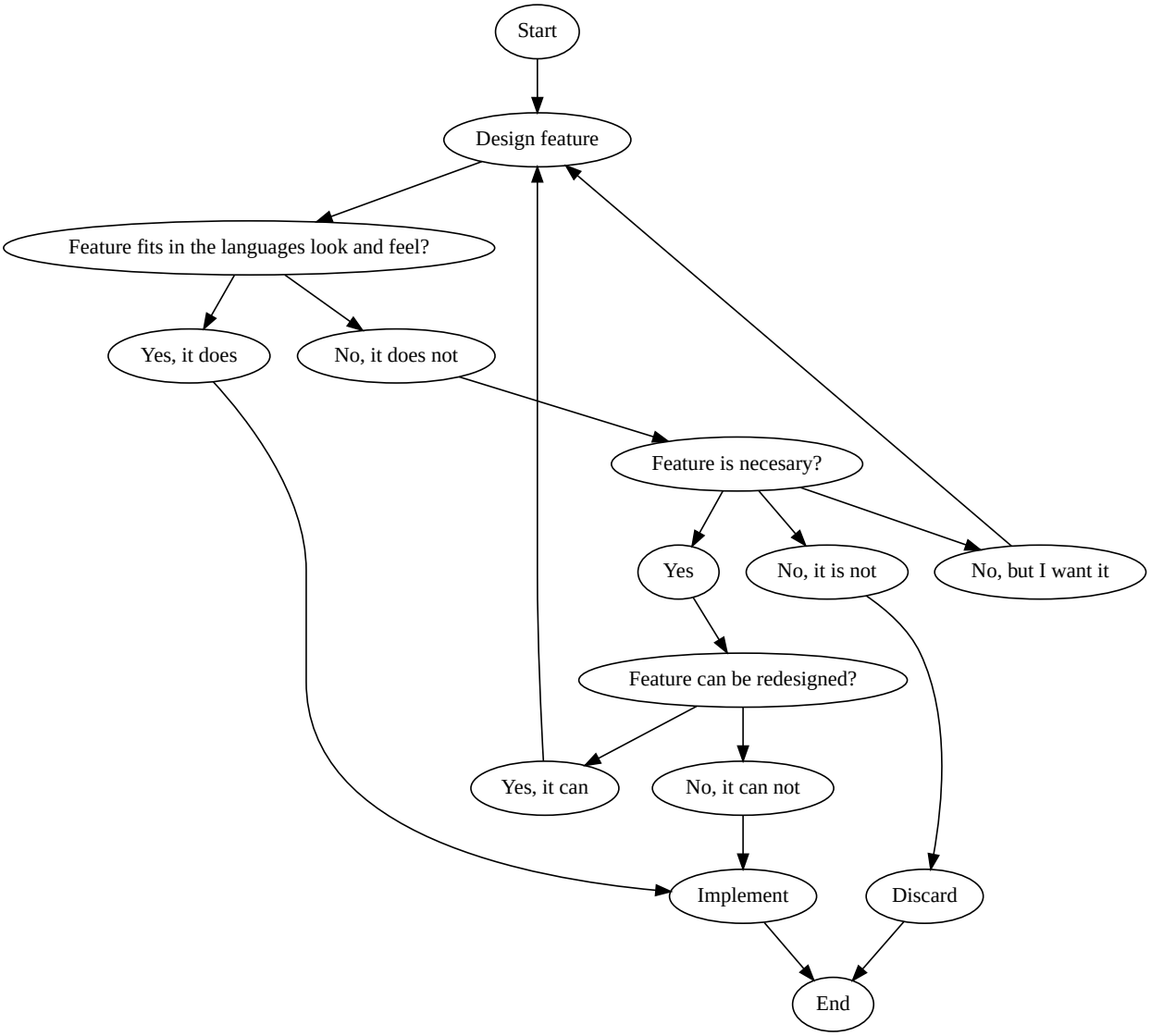
```
statements = { statement } ;
statement =  pipeline | ... ;
pipeline = statement, "|>", statement ;
```

With our EBNF ready, we need to write a code sample with our feature as the focus.

```
Console.ReadLine()
    |> File.ReadAllBytes
    |> SHA1.Create().ComputeHash
    |> BitConverter.ToString
    |> Console.WriteLine;
```

Implementing a feature

Because we want to avoid adding unnecessary or half-baked features, we need to first think about really adding a certain feature. This keeps us from losing focus on what's important and adding the wrong features to our language.



Flowgraph for implementing a new language feature

For our new C# feature, this is fairly trivial. Our new language feature does, in fact, fit in the language's look and feel. Therefore, we can implement it.

If it didn't fit, we'd have to ask ourselves if the feature was necessary and if there is another way to solve the problem we're solving with this feature (even if said way was syntactly less pleasing). If that were the case, we could either discard the feature or redesign it.

If the feature was necessary, and can simply not be redesigned (this mostly happens because of other design flaws in a language), we'd have to implement the feature to our best of knowledge and belief.

Writing our own mini language

We will be writing our own procedural/semi-functional language. It will be called "littl" and its file extension will be ".lit". We will transpile the language into JavaScript.

Definiton

The language isn't really all that powerful. We have variables, which we can declare with either `var`, for normal variables or `var!` for constants and there is shorthand declaration with the `:=` operator (which doesn't require a var keyword). We won't deal with datatypes but rather let JavaScript handle that.

Littl also has C-style comparison and mathematical operators. Values can be returned with the `return` keyword.

Functions

Declared like: `functionName argument1 argument2 ⋯`

Example

```
add x y {
    return x + y
}
```

For loops

Declared like: `for variable in array`

Example

```
arr := {1,2,3,4,5}

for i in arr {
    console.log(i)
}
```

Counted for loops

Declared like: `for variableDeclaration; condition; operation`

Example

```
arr := {1,2,3,4,5}

for var i = 0; i < arr.length; i = i + 1{
    console.log(i)
}

for ;true; {
    console.log("Hello")
}
```

If condition

Declared like: `if condition`

Optional: `else`

Example

```
a := 5

if a < 10 {
    console.log("a is smaller than 10")
}
else{
    console.log("a is bigger than 10")
}
```

These statements have C-like blocks (curly brackets).

Because littl transpiles to JavaScript, one can use JavaScript functions like `console.log`.

A recursive fibonacci sequence would look like so:

```
fib n {
    if n is 0 or n is 1 {
        return n
    }

    return fib(n-1) + fib(n-2)
}

x := fib(12)

console.log(x)
```

Since littl is part functional, every scope, be it `if`, `for` or a function scope, can be put into a variable or returned.

```
count := for i := 0; i not 10; i = i + 1 {
    console.log(i)
}

count()

var! two = 2
twoSmallerThanThree := if two < 3 {
    return true
}

twoSmallerThanThree()

curryAddition := anonymous x {
    return anonymous y {
        return x + y
    }
}

add3 := curryAddition(3)
console.log(add3(2))
```

Setup

For this example, I'd recomend using a UNIX based or unixoid machine. I'll be using Ubuntu 18.04.1 LTS 64bit for both the setup and the programming part. My editor of choice will be VSCode with the following extensions:

- Lex Flex Yacc Bison [16]

  and

- C/C++ [17]

Packages

```
sudo apt install flex bison make gcc g++ -y
```

Lex/Yacc hands on

We could write our lexer and parser ourselves but that would be too much work. Therefore we utilize Lex and Yacc. Lex and Yacc are definition languages that serve as a high-level EBNF. They provide a stable framework to build any form of language runtime on.

Flex and Bison are GNU implementations/extensions to this framework.

Our Flex file serves as the lexer. It will get a filestream from the file we're trying to transpile and convert the content of said file into a stream of tokens.

The Bison definition will be the parser. It will receive the tokens from Flex and parse them by recursive descent.

Since Flex and Bison run on C and C++ respectively, we will be writing our transpiler in C++ as well.

lex.l

```
%option noyywrap ①
%option yylineno ②

%{
    #include "grammar.tab.h" ③
%}

④
INT                             \(-[0-9]*\)|[0-9]*
DECIMAL                         \(-[0-9]*\.[0-9]+\)|[0-9]*\.[0-9]+
STRING                          \"[^\n"]*\"
BOOL                            true|false

NAME                            [a-zA-Z][a-zA-Z0-9._]*

VAR                             var
CONST                           var!

IF                              if
ELSE                            else
FOR                             for

...

⑤
WHITESPACE                      [ \r\t\v\f]
FEED                            \n

%%

⑥
{STRING}                        return STRING;
{INT}                           return INT;
{DECIMAL}                       return DECIMAL;
{BOOL}                          return BOOL;

{VAR}                           return VAR;

...

⑦
{WHITESPACE}

{FEED}                          yylineno++;
%%
```

① Disable `yywrap` - after an EOF, the lexer assumes that there are no more files to scan

② Expose `yylineno` (current line number)

③ Import header file of grammar

④ Define tokens

⑤ Define whitespace characters (these are ignored) and linefeed (so that the lexer knows when to increment `yylineno`)

⑥ Return enum for each token

⑦ Ignore whitespace and increment `yylineno`

grammar.y

```
%{
    #include <math.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include "../src/tree/nodes.hpp" ①
    #include <iostream>
    #include <vector>
    #include <memory>

    extern int yylineno; ②
    extern char* yytext; ③

    int yylex(void); ④
    void yyerror (char const *str) { ⑤
        fprintf(stderr,"Error | Line: %d\n%s\n%s\n",yylineno,str,yytext);
    }

    ⑥
    littl::SyntaxTree* root;
    littl::SyntaxTree* p;
    std::vector<littl::SyntaxTree*> tree;

    #define YYSTYPE littl::SyntaxTree* ⑦
%}

⑧
%token INT
%token DECIMAL
%token STRING
%token BOOL

%token VAR

%start input ⑨

...

%%
```

```
⑩
input:
    program { root = new littl::Program(tree); }
    ;

⑫
program:
    block program { tree.insert(tree.begin(),$1); } ⑪
    | %empty { $$ = new littl::Empty();}
    ;

⑬
block:
    assignment { $$ = new littl::Terminated($1); }
    | statement { $$ = new littl::Terminated($1); }
    | variable { $$ = $1; }
    | return { $$ = $1; }
    | function { $$ = $1; }
    | if { $$ = $1; }
    | for {$$ = $1; }
    | block block { $$ = new littl::Tuple($1,$2); }
    | %empty { $$ = new littl::Empty(); }
    ;

...

⑭
statement:
    name LBRACKET RBRACKET { $$ = new littl::Call($1,new littl::Empty()); }
    | name LBRACKET parameters RBRACKET { $$ = new littl::Call($1,$3); }
    ;

...
```

① `../src/tree/nodes.hpp` contains a list of all sub-node classes

② Reference `yylineno`

③ Reference `yytext` (refers to the actual string value of tokens)

④ Reference `yylex` method so that the parser can receive a token stream

⑤ Define error function

⑥ Define root node and temporary nodes

⑦ Define the datatype of the base node (every other node inherits from the base node)

⑧ Declarate all token types used in the lexer

⑨ Declarate the starting point of the parser

⑩ Starting point of the parser

⑪ A program can contain multiple blocks

⑫ `$1` refers to `block` (the first item in the rule)

⑬ Defines all possible block types

⑭ Sample rule

tree/syntaxtree.hpp

```
#pragma once
#include <vector>
#include <string>
#include <memory>

namespace littl {
    class SyntaxTree {
        public:
            SyntaxTree() = default; ①
            virtual ~SyntaxTree() = default; ②
            virtual std::string toCode() const = 0; ③
    };
}
```

① Constructor

② Destructor (we will be dealing with many objects, we have to delete the memory used by them properly)

③ toCode function, generates code from a node

Here is a sample for how a node could look like:

tree/blocks/if.hpp

```
#pragma once
#include "../syntaxtree.hpp"

namespace littl {
    class If : public SyntaxTree{
        public:
            If(SyntaxTree* condition, SyntaxTree* block){
                this->condition = condition;
                this->block = block;
            }
            virtual ~If(){
                delete condition;
                delete block;
            };
            virtual std::string toCode() const{
                return "if(" + condition->toCode() + "){\n" + block->toCode() + "}\n
";
            }

        private:
            SyntaxTree* condition;
            SyntaxTree* block;
    };
}
```

main.cpp

```
#include <iostream>
#include <fstream>
#include "tree/nodes.hpp" ①

extern int yyparse();  ②
extern int yylex(); ②
extern littl::SyntaxTree* root; ③

int main(){
    int result = yyparse(); ④

    if(result == 0){
        std::cout << "Input is valid!" << std::endl;
    }else{
        std::cout << "Input invalid!" << std::endl;
    }

    std::ofstream outputfile;
    outputfile.open ("out.js");
    outputfile << root->toCode(); ⑤
    outputfile.close();

    return result;
}
```

① `tree/nodes.hpp` contains a list of all sub-node classes

② Reference to methods exposed by Flex/Bison

③ Reference to resulting root node exposed by Bison

④ Lex & parse the source file, result is put into `root`

⑤ Generate code from root node

The entire code for this transpiler can be found on GitHub.

littl [https://github.com/Azer0s/littl]

Conclusion

Creating programming languages isn't as complicated as it seems. With the right tools and a bit of know-how it becomes fairly trivial.

The last hint: listen to what users say they want. Don't implement what users want (you are the language designer - not your users). Implement what you think is a good solution to their problems. Giving your users a better tool to solve their problems than they could've thought of is the ultimate test of success in language design.

> The test of success in programming language design is being smarter than your users. This is also the case in programming methodology. Perhaps these two are the same subject anyway.
>
> — Unknown

References

- [gcc] GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF): https://gcc.gnu.org/
- [roslyn] dotnet/roslyn: The Roslyn .NET compiler provides C# and Visual Basic languages with rich code analysis APIs: https://github.com/dotnet/roslyn
- [javac] javac - Wikipedia: https://en.wikipedia.org/wiki/Javac
- [babeljs] Babel · The compiler for next generation JavaScript: https://babeljs.io/
- [coffeescript] CoffeeScript: https://coffeescript.org/
- [emscripten] kripken/emscripten: Emscripten: An LLVM-to-JavaScript Compiler: https://github.com/kripken/emscripten
- [v8] V8 JavaScript Engine: https://chromium.googlesource.com/v8/v8.git
- [hhvm] facebook/hhvm: A virtual machine for executing programs written in Hack: https://github.com/facebook/hhvm
- [cpython] python/cpython: The Python programming language: https://github.com/python/cpython
- [pypy] PyPy - Welcome to PyPy: https://pypy.org/
- [cling] Cling: https://cdn.rawgit.com/root-project/cling/master/www/index.html
- [turing1] Turing, A. M. (1936). On Computable Numbers, With an Application to the Entscheidungsproblem [https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf]
- [goedel1] Gödel, K. (1931). Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I [http://www.w-k-essler.de/pdfs/goedel.pdf], Monatshefte für Mathematik und Physik, v. 38 n. 1, pp. 173–198.
- [currying] Schönfinkel, M. (1928). Über die Bausteine der mathematischen Logik [http://www.cip.ifi.lmu.de/~langeh/test/1924%20-%20Schoenfinkel%20-%20Ueber%20die%20Bausteine%20der%20mathematischen%20Logik.pdf], Mathematische Annalen, v. 92 n. 1, pp. 305-316
- [r] R: The R Project for Statistical Computing: https://www.r-project.org/

- [lexflexyaccbison] Lex Flex Yacc Bison - Visual Studio Marketplace: https://marketplace.visualstudio.com/items?itemName=faustinoaq.lex-flex-yacc-bison
- [cppcode] C/C++ - Visual Studio Marketplace: https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools