FORMULATING THE SYNTACTICALLY SIMPLEST PROGRAMMING LANGUAGE

Brendon Boldt
Language Technologies Institute
Carnegie Mellon University

ABSTRACT

The syntax of a programming language is one of its most visible characteristics, and thus, heavily shapes how users both interact with and view the language. Abstruse syntax correlates to lower usability, and therefore, lower adoption of a programming language. In light of this, we present a programming language with the simplest possible syntax. In addition to the language specification we give a basic style guide and reference implementation.

***K*eywords** programming languages · compilers · formal grammars

1 Introduction

As computers become further and further prevalent in modern day society, computer literacy similarly grows in importance. Taking the analogy further, composition is to literacy as computer programming is to computer literacy. A novice programmer will often find it difficult to address the syntax of a programming language at first, which hinders the acquisition and fluency with the semantics of the programming language [6]. In order to lessen this barrier of entry, we will explore a programming language with the simplest possible syntax.

The complexity of the syntax of a grammar is dependent on a number of well-defined factors: number of rules, cardinality of unique terminals (tokens), and number of terminals and non-terminals per production. These are all further characterized by probabilisitc factors, namely their distributions and associated information-theoretic entropies. Determining the equivalence of two grammars is an undecidable problem, and the computation of the entropy of grammars and their syntax trees is an open area of research [7]. Thus, these considerations are beyond the scope of this paper. Regardless of the formal specifications of complexity and entropy, we assert that fewer unique tokens, fewer rules, and lower branching factors all correspond to simpler syntax.

The core concept of our programming language is that of simplicity. Fittingly, the name we have chose for the programming language is "SimPL" standing for "Simple Programming Language".[0] We present the formal specification of SimPL 2 in Section 2.[2]

1.1 Related Work and Scope

SimPL bears some resemblance to so-called "esoteric" programming languages or "esolangs." Such programming languages typically try to maximize goals, such as minimality and creativity, that typical programming languages typically leave by the wayside. Two well-known esolangs are brainfuck and Unlambda which are a Turing tarpit and Church quagmire respectively [2]. Such tarpits and quagmires seek to limit the semantics of a programming language to the bare minimum as a demonstration of how little is required to be computationally universal.

Another direction of programming language minimalism comes in the form of *one instruction set computers* (OISCs) [1]. These are specifically *machine* languages which provide only one instruction (operator) which, in turn, can take

[0] This is not to be confused "SymPL" or "Symbol Programming Language" which is the true name of APL.

[2] The astute reader will notice that we have skipped straight to SimPL 2 (typically, 1 comes before 2). For an explanation of this, see Footnote 1 (although 2 has preceded 1 in this case).

variable arguments. While OSICs and SimPL both center around the idea of “unity” in terms of representing a program in a language of some sort, SimPL does this at the level of a programming language while OSICs are, by definition, concerned with instruction sets.

While the aforementioned languages are mostly interested in testing the limits of minimalism in computability (either through limited semantics or instruction sets), our work adheres to area of language design purely with respect to syntax and syntactic usability. Thus, rather than introducing a language with entirely new semantics as well as syntax, we will recycle familiar semantics packaged in a novel syntactic model. In this sense, our language bears similarities to transpiled languages such as CoffeeScript or Dart.

2 Language Specification

2.1 Syntax

We begin with the Backus-Naur form of SimPL.

$$\langle start \rangle ::= \langle expr \rangle$$

$$\langle expr \rangle ::= \langle tok \rangle \langle expr \rangle \\ \mid \langle empty \rangle$$

Note that there is only one lexical token SimPL, which can be represented an arbitrary charcter, emoji, or any other kind of thing. While we give the BNF of the grammar here. We have artfully crafted the syntax such that it does not need to be expressed as a context-free grammar (Type-2 on the Chomsky hierarchy). We can, in fact, express the syntax of SimPL with a finite state automaton (representing a Type-3 grammar). While this specification seems small, one could imagine it smaller, but we ran into serious problems attempting to use a simpler specification than presented above.[1] For example, a string can be determined to be within the grammar using the following regular expression (Perl compatible):

```
.*
```

Figure 1: Perl-compatible regex specification for SimPL.

2.2 Semantics

The semantics of a programming language are concretely expressed in machine code. This transformation is achieved through the compilation of a programming language. The relationship between the source code of different lagnagues as well as their machine representation can be represented as a simple subcategory of **Set**. Namely, our objects will be the set of source strings for a given programming language with a special object for the set of all machine code representations. Morphisms from source code objects to machine code objects are realized by the process of compilation. If we take every programming language to have a canonical compiler implementation, assume that decompilers do not exist, and consider only one machine architecture, we can view the machine code object as the terminal object of our category.

We will use S_C and S_S to refer to the objects of the C and SimPL languages respectively and M to refer to the terminal object corresponding to machine code representations. $k_C : S_C \to M$ and $k_S : S_S \to M$ are morphisms that correspond to the compilation of a programming lagnauge. While both GCC and Clang could be seen as different morphisms from S_C to M we would consider the languages to be distinct since the same source strings correspond to different machine code representations.

In this paper, we focus on simplicity of syntax, and in order to keep other factors constant, we tie the semantics to a well-established reference point, namely C. To express this more succinctly, we will introduce the term *semantic equivalent in C* or SEC. A SimPL source code string corresponds (semantically) to a unique C source code string—this C source code is the SEC of the SimPL source code. We formally define the semantics of SimPL as $k_S = k_C \circ r_S$ where r_S is one component of an isomorphism between semantically equivalent C and SimpPL source code. The components of this isomorphism are $r_C : S_C \to S_S$ and $r_S : S_S \to S_C$ such that $r_C \circ r_S = \mathrm{id}_{S_S}$ and $r_S \circ r_C = \mathrm{id}_{S_C}$. This is the illustrated by the commutative shown in Figure 2.

The expressivity of the syntax of C an SimPL may seem too widely disparate to be practical, yet we can actually define the isomorphism between C and SimPL source code. In particular, we will call this isomorphism *radix representation*

[1] See Appendix A for specifications that are not 2 SimPL.

$$
\begin{tikzcd}
S_C \arrow[r, "k_C"] \arrow[d, bend left, "r_C"] & M \\
S_S \arrow[u, bend left, "r_S"] \arrow[ur, dotted, "k_S = k_C \circ r_S"'] &
\end{tikzcd}
$$

Figure 2: Commutative diagram illustrating the category-theoretic definition of SimPL's semantics.

`00000000000000000000000000000000000000000000000000000000000000000` (a)

`ROhNGKVUZH0cQvoy5jBG0tFLNkeFPS9c07oGlL5D0jVlU3l1M2WxQcggM1x3VKv3e` (b)

``1\8+mm;/Q9F1e:hs1+SP1xQLW3&[Mh?ohJMX47zaGO~?3Vp‘maU’onby(i,^p_r#,`` (c)

Figure 3: Assorted SimPL source strings for the SEC `A`.

mutation (RRM). As C can be represented as a sequence of ASCII-encoded bytes, We can express any C program as a radix 128 number (as standard ASCII values can be represented with 7 bits) with each numerical place corresponding to an ASCII character in the C program. SimPL, on the other hand, is represented by a radix 1 number. Thus, we can morph C source code into SimPL as follows.

$$\sum_{i=0}^{n-1} c_i \cdot 128^i = N \to 0_0 0_1 0_2 \ldots 0_{N-1} \tag{1}$$

Similarly, given a SimPL program of length N, we can generate the SEC where the zero-indexed ith character corresponds to:

$$\left\lfloor \frac{N}{128^i} \right\rfloor \bmod 128. \tag{2}$$

2.3 Syntax vs. Semantics

It is worth addressing briefly a common objection to our delineation between syntax and semantics. The objection is that SimPL's syntax is simply a trivial shell and that the real syntax (that of C) is masquerading as the "semantics" of SimPL. Hence, the true syntax of SimPL is just as complicated as that of C. One illustration of the objection consists in the fact that SimPL has no "syntax errors" *per se* but instead mutates C syntax errors into SimPL "semantic errors."

In response, we first point out that there are many different levels of errors beyond syntax and semantic ones. In roughly ascending level of "depth" we have: lexical (errors), syntax, semantic, logical, and design-level errors (though this list is not exhaustive). Although these levels are not strictly defined, one of the sounder heuristics for determining the level of an error comes from looking at from which stage of the compiler the error comes. The fact is that syntax errors in the corresponding SEC would be generated by the compiler backend after parsing and AST generation, and thus would be out of the scope of true "syntax errors."

Furthermore, in some cases there is not even a clear distinction syntactical and semantic errors. For example, in Python, using `return` or `yield` outside of a function yields a `SyntaxError`. Although trying to return or yield from outside of a function is a *semantic* error rather than a syntactical one, this difference is not maintained in Python [3]. Further debate as to the precise distinction between syntax and semantics could provide useful topics for future work but is beyond the scope of this paper.

3 Representation

3.1 Naïve Representation

The naïve representation of SimPL entails representing each token as a single character. For example, representing the SEC `A` in SimPL could take the various forms specified in Figure 3. It becomes evident when we give the naïve representation of the SEC `AB` that this mode of representation quickly becomes unwieldy; for formatting purposes, the SimPL source string has been rendered in Appendix B.

As is the case with all programming languages, syntactically correct does not imply readable or maintainable. Thus, we present some potential stylistic improvements in order to reduce cognitive load when determining the number of

```
.........|.........|.........|.........|.........|.........|.....    (a)
.........10........20........30........40........50........60....    (b)
.........ten.......twenty....thirty....forty.....fifty.....sixty.    (c)
12.4...8.......16..............32..............................64    (d)
...............................................................65    (e)
```

Figure 4: More easily readable SimPL representations for the SEC `A`.

```c
#include <stdio.h>

int main() {
    printf("%s\n", "Hello, arXiv!");
    return 0;
}
```

Figure 5: A basic "Hello, arXiv!" program in C.

characters in Figure 4. The choice of using delimiters every 10 is somewhat arbitrary. This is certainly appropriate for situations where the program length is not many orders of magnitude greater than 1 but could be counter-productive otherwise. Thus, we could turn to constant multiplicative spacing between delimiters instead of constant additive spacing.

While these intermediary delimiters can make it easier track one's place in the source code, generating such delimiters goes beyond what is strictly necessary. For example, we can simply express the number of tokens at the end of the program.

3.2 Practicality

In a more practical example, take the basic SEC of a "Hello, arXiv!" program show in Figure 5. The above program consists of 86 characters; thus, the naïve representation consists of $(2^7)^{86} = 2^{602}$ tokens or about 2 tebiyobiyobiyobiyobiyobiyobiyobiyobibytes (if each token is represented by 1 byte). More generally, an n-character SEC requires $(2^7)^n$ bytes to represent.

At very large-scale code bases, simply comprehending the scale of the representation (let alone actually representing) becomes difficult. Take the Linux kernel, it has on the order of 2×10^7 lines of code [5]. If we use the estimate of an average of 40 characters per line, that gives us 8×10^8 characters. The SimPL representation of this would, then, require on the order of $(2^7)^{8\times 10^8} = 2^{6\times 10^9}$ tokens to represent. Any explicit representation of this number of tokens surpasses any current or theoretically possible information system. Although this exhibits exponential growth, this number is not very large in the context of mathematics, being $2 \uparrow\uparrow 5 < n_{\text{Linux}} \ll 2 \uparrow\uparrow 6$, but this number is still large enough to make pursuing more efficient modes of representation prudent. Thus, while this mode of representation works well for explaining the conceptual grounding of SimPL, a more efficient mode of representation is needed to effectively store and process SimPL.

3.3 Compressed Representations

Let us revisit Figure 4e, namely the representation which consists of a repeated, non-numeric character followed by the number of total tokens at the end (expressed as a radix 10 number). By observing this style of representation, we can see that number at the end by itself (i.e., without the repeated leading characters) could serve as a sort of shorthand representation of the whole program. This sort of shorthand makes the task of representation far more tractable. For example, simply using a radix 10 representation for the "Hello, arXiv!" SEC would as presented above would yield $\lceil \log_{10} 2^{602} \rceil = 182$ characters in total.

Using a higher radix could give us an even more compact representation; for example radix 64, common in text-based data transmission would give us $\lceil \log_{64} 2^{602} \rceil = 101$. If we take this even further, we could use the cardinality of Unicode 12.1 characters which stands at 137 994, which leads us to an even more compact representation $\lceil \log_{137,994} 2^{602} \rceil = 36$. Although, this becomes unwieldy not by virtue of its length but on account needing to have complete familiarity with

```
#include <stdio.h>

int main() {
    printf("%s\n", "Hello, arXiv!");
    return 0;
}
```

Figure 6: The Canonical SimPL of the SEC shown in Figure 5.

every Unicode character. Thus, the optimal representation will strike a balance between compactness and recognizability of the set of characters used.

"Canonical SimPL," as we call it, uses a radix 128 representation of an SEC where each digit is rendered as its ASCII equivalent (e.g., 65 as `A`). An immediate issue seems to arise from the fact that there are 33 unusable values (either non-printable characters or unmapped values) in the ASCII encoding [4]. Yet due to RMM with C, any semantically valid Canonical SimPL source code will only ever correspond to printable ASCII characters. In this way, just as RMM provides us with an isomorphism at the machine-interpretable semantic level, Canonical SimpPL provides us with a sort of isomorphism at the human-interpretable level. As an illustration, we have shown the same "Hello, arXiv!" program written in Canoncial SimPL in Figure 6.

4 Reference Implementation

In our reference implementation of SimPL, we limited our scope to compiling Canoncial SimPL source code. This compilation, in fact, can be done entirely with a typical *nix toolchain. For example, for any given Canonical SimPL program `lorem.spl`, the compiled binary can be generated as such:

```
cp lorem.spl lorem.c && gcc lorem.c -o lorem
```

This represents the GCC dialect of Canonical SimPL; the Clang dialect would implemented similarly.

Acknowledgements

Special thanks Dr. Bartosz Milewski for informative YouTube lectures on category theory.

References

[1] In *Esolang*. *https://esolangs.org/wiki/OISC*

[2] In *Wiki*. *TuringTarpit*

[3] Guido van Rossum et al. 7. Simple Statements; 7.6. The return statement In *The Python Language Reference https://docs.python.org/3/reference/simple_stmts.html?highlight=syntaxerror#the-return-statement*

[4] W3Schools. HTML ASCII Reference In *HTML Charsets https://www.w3schools.com/charsets/ref_html_ascii.asp* June 16, 2019.

[5] Linus Torvalds. Linux kernel. In *https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/. commit hash 0ee4e76937d69128a6a66861ba393ebdc2ffc8a2* (from "master" branch on June 5, 2019).

[6] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *Trans. Comput. Educ. 13, 4, Article 19* (November 2013), 40 pages. *DOI=http://dx.doi.org/10.1145/2534973*

[7] Werner Kuich, On the entropy of context-free languages, *Information and Control, Volume 16, Issue 2, 1970, Pages 173-200, ISSN 0019-9958 https://doi.org/10.1016/S0019-9958(70)90105-1.*

A Earlier SimPL Iterations

SimPL 0 was described by the empty grammar. Though the syntax was very simple, the semantics of the program were static, that is, SimPL 0 could describe one and only one program. We found these semantics to be unreasonably limiting. Thus, we decided that a non-empty grammar would be necessary.

$\langle start \rangle$ ::= $\langle tok \rangle$

Simple 1.0 introduced having at least one rule in the grammar. Although it now conceivable that program could be written according to the grammar (namely a single token), there is still only one possible program. In this way, SimPL 1.0 is equivalent to SimPL 0. We proceeded to Simple 1.1 as such:

$\langle start \rangle$ ::= $\langle expr \rangle$

$\langle expr \rangle$ ::= $\langle tok \rangle$ $\langle expr \rangle$

Although programs can now consist of multiple tokens, the only valid program is in fact one consisting of an infinite string of tokens. In this way, SimPL 1.1 is equivalent to SimPL 1.0. Finally, we added the grammatical option to terminate a multi-token program which we present as SimPL 2 as described in the body of the paper. In short, SimPL 0, 1.0, and 1.1 are too simple but 2 SimPL is not.

B C Source Equivalent of `AB`


```
iwiNLHDAdMgA7hrgfoM6BNyYLXJYBoY916TUl1FdKL2uFRprQRJ7O3M8EbXfw3KK99dZE5Aby3GTQGtvLf6LXDr9O
AbDXjd367WY47Zra42aAvKMOaW3bA7WjyU1vnhV6LdwUmjCNB9dOq4RhfPnJ0KDe9fwMuPg78AMK4UT8VomZGUu9Z
CmeyoWo5OGNRVclcxVwVVhuJdj0lVuJuyAGZPzsayD8g7FhX6Gug2tHYHDblNQaoTCjKYpDa4Hx245qRneGLy7pDE
19rW6gtW1dgMQK3trBU7E69jSs6hXPqKBgMAdDtiaW5i8N5of91JopS7lqEaScxPGrfVxqEdrcfyD8mNyuFicrego
kiTQCVScfQW4OznWZuBbZCnI8kiZS91pWpMGEo6aydmw0A5A8CWRczpFBCjLvSd6NHIFwXoNkplmBIOyzG3d0Girs
2biZjHylBbZacd4BIp4P21zOp6qmt9O9K7T1ovEwfeFiwtPk0lFm5OJJkDCouNgZoBYJL1bEAMuo2C4q4cpwfU5YO
AfxUvv5tzjCPDxuev6jMMlPfdzib0SAs9i9lRInkqen7MCFIXYdtS8MsZvNfW5ZU0tTKrmUaoVlAlo19pQecjPgFP
SZg4W4TWILFAAuckUjOUh7dmcSNFWE6VLaIhbZVPA9iRmIuj03KKG1L8HaF6LtBXu8Et0m3pHPNjQ98u3HemaWwse
VMqKGwBEXwTzksVcBUQRUUZHh9dN9yhs4bF2DCdogb2suzRb5btMBSVltu1T2EO3THzmu6IebEmL7gVvvYmQw0frk
RTreSj5IoHT5gr13XwH8GB73DIh5l14n77CmDzCQCIhQJvdRKA1fR1mDPypEai1hmC0qeHYEoG8CpdT5Rah66DWva
ou6hEqrsxQXvxYZ627Mn1heglNz205BgaJI4b1WWM5gs1VALJ5UZJHm0bSEqbizP3Glszq4JDmBIjDdJQfH0cDlLX
LsEzi1s2xMyj8LR6KgkGRlXLrbN2RVSfsADadMqsguERxxMUzlwNALOg6Ev2noQRNJ7pSkQaFVI7yDt3J84dWXY8A
qE9J8jbyVqAbdAeRsz34UIUuukXMAynFP1PB1O9PQ7DMS0gn5VDLoHwfbLvc5ngaD6KiST1MLVYVEHjnuLIo3oFAt
RrqBTFlNDnr2ehsGmZOGQ5OqjniUAG6oeixbn8AF1Pfh4Feg8zNP2MZqOjViEar8ZZaPTil4NuZt5NHWhMzjUdIST
ATiijNXUmXxCMhZrxgHSd4RHtDLKfn3f7oYGRuSNU8NOnR8M0kHLlbjl8IvxTNGmLL8cUovBEquYcDKL5V1vklwQg
aaP5CjKLoqANY1SsPTrLw5DJKXTqJDlC4u8HzNiJTT8tAHyUVM6uSXuUOhTFpFhNgrwNAwROIsgYKNUMG4Ak2dY94
vCuh3kPxTXu4akOMFDTnqfn8zod0Zhj82qiNBcfeq3mskjJTeBrjklRTPo4b1jdznouI6DoFn27CHeWGCuzbvfpuI
Wne4XFCkEk8V0ZXuPh4qSgHLTLBFVMwAU5ATzqO6zYyN6jx62RVmWHlUBIqsHM6cMSojSvT0rP95HTy7d6PdPQrUB
GlM3Og4LYqYbYQO1KaDa5vjvBwHIo4SucjbCmbWw4N12DqsxaTuVpVq6MHuHQ7aoJxvBFs176j6D3Hz1Ongr9aHF8
OtJWR11rqNsgvIPRsdfMKtBPKwa5yXE0IYAk9vwhZCws6LL8hZQzEwAlisUm2Lqe4Jj8zAV6ybTa7SKBFcj8vRDjv
ITyciIK79ZX3S32PMS7AJTtonE8iEjcHxuMLCseB8w0Zwwm4FnPqnb6UtO9tPNZhdoyGu1UBzURVnYWu8HLqsT5sw
2h91ugywjhp63sJUcGJh4lXFPQ1AjEnhLF2s2dIdMLEZMYJMDzUDLbewuk4xTxH6Wg31sQCFWUyuKVLpi2Q3EK62x
kQeBMJH3XOFSkDOdvOkSO3TZPBmzDrUmmrxmyjro9LoVNGnllXDZXFVmHeK5ux5cXr7VMF4mW1kf9TfXReLf31B5W
LFELxnRSE8ptpb8TtIHJOIVQaaLMNeBUDdqejSPQ0i0BXdYasLabblBm1O16BPJ0nlNO31JLnbe6rHmGtOkvlm1Fl
4rGjI86vdZE58Qr6Ej4edaF6aQ6XjRy0tNfYLDUJ26Gzm1haMDjToAYyyh8ccIJleBTpw9D21EAeYaOK6ERhfxMXi
qzBvyMrYMJben7uUmcGOCbDYIAWHZbmPu5eOrkXOSjc6snOsQPUCALcBnFs3BMZsA40BFXr68Rq3OgRCkN6MXp5HV
9AkIbWcNJGAIDOeUSiNSwTgZwnMVixqbCeaDJcyW0ftereVahD6Qzzicxtba4z6e8iwuoSiWJ0FJ5yuGeEasxYXec
XDMCky7mCLXsajHAyT7RafcB18klOa9AFccQCV9P1AXzGTOpEm7fHC4Jz8AoqDW5IOzAihVcz8vpsT3oijH6UWM9B
3AbSExDFnu8L63dj98d8LxiPgbGoOqEj6sbkkzZfMpE0fbU0HiXDIsgpLhB1MhTPWy8uq0va2JVsCcBGBA661Md8u
BdCwFnPltRlZv1Cydc4BoNEY7mvJwbiRiu6pKcyHi5lBHryhSZGacpa2jMBvoF3FKuJFZItGNkuTttELdkGaZiJeD
xKHGOjS8FP0PGR6eTJfNGmqg88YhD8uJv1QM8bdKpwtnGU3Uw3o3Ywi3eyAEOYb7jeaAM1CgiMvMSf55HNO6GW6Ce
CUW97LzqBD4LqZ28uzi9rf87aXpb5yV2oTsqbbjeHykRujW3SGBobfIjAEvbm7mFd8GzNgVD89hVihSMEJseS18zz
Pk4wCOa4bV81AMxbxsFTdMf0ZJ9CqyMRG7WwE53zNN5kiYGt9oRcsHexYDFhbAtqOVQsxyIwCoUjD6K8Zv64WhrzX
LhHQu5iDa6BR5s6WmUtGQfm7OY9My5YUBZi1alHSRFCRT5MGJ5yCt6NaLsg6iRIFpeMa1i1w7YjjbB48txKBB2Lvw
kKNqlFwUXnbjXIf1mPRuQiPhNR0DLSA7vXmCNe4LQm5AHYphj6jzbL06MnuxmehSKbNVwKWE0paM768xVOQ82ejzH
CP3eUZKrKeClF0t2ONf6kqW2dC35PqhrmKeD6lvuBdYNSGqhcXI1PRTc8TAoEtgxGefzgsW6TFiP7qq4DnhC5yyJd
CFa6Hq0UBJjLJgw3dR09GB81TXqTOabvhXJPSNkg3joVH7RNQyGFLPBvDGvccQygRHGgPURhIt9O9N8nkp21YAe7u
2CKuhTHXe7ntXjXEcWNNAUtRX5IRN7sjVUUhYmpsvBIMPvJEfTEUTm6s7pI0nOpdempLPlN1S7JKszNCcxA1747Ax
JzsoOYUX0PuJTi9qSqKywZ1ou6NI8Y4KgmFqPrOln05qumlxkELWGlahKG2k1bE5Kswzo21Obpk7Ya1lhho83qNqD
RMakqHhoXtTzTqBKX7YqCetWOq4ULn8FgnBbWEMgAWjXXiDbFySezSusRIkPp5Jxa5T1X3TI3jGe5fqK1UewBEDxx
Y41fNjHUHFyKdDPfKcJvygva7Tywb8noeotf8CTrrfqojS3LBjrygeWmu97s31qYg2ZCYddYkC9jiAH7OhqCaKBMn
z7T1HJKQYJeT5ZA1w7fI5KqQpAimBWKHEKaDqHBPiqPY6dvYwfA78hNdsH7HGQSHYFH9UwiX9Vqu84OyCEQSKdctJ
UQFkcd6cleB8FSMj7EsjykKKA6koioHTZduyUfiAgJffAasDeGXDekD6SlrZU3phWU5QgnrJUvEZakAAQgrOiTvnD
```

hHAoNS4DMrZNOzOPDoQlgYj7OZKisvKmZ25cfwfh1F0qkHrQAaqojHz2b1yj0MgCc1YuybMiRgrBjT1Q9GZnJVgqB
cyWRJ9CryzM6TP4UiYxi5je29vjnevXrwqPjkKftYk1UzbPX8Tbr4bDv25vMGycQEsspOB3PNeOrqaIcRtYM3AirB
UROVOBzI3AUOuJI71lEaCU9y6NwJuhKWUjVAiTA6ncb9vg2Ok30h9ojCnmV15aEiPLqixjeJt0Yg9pHCj124INZUH
jUBEFEb8yTIJFufHxQVB1XMwZFAQLFk48t4p16e6E2kASqggtdoxfyv7lAD83bQZ3gMJefcvAjM6CmghyjPjMipTY
XhscKk4dTQFYLIwqFINGWuJqJYwSoALO0qLQFZnyWv1BJiV7enDf3kEQXA3lNFuqfLdpVxBOIYbrsuSWtFlVI5gti
F29VuYarLAMSrnD4vHvxLxvClfJARmEnvdpIWgaNAoI8vYFoanQgsYUL5axFHSGdZclI6hXAIf9Wy7fC2DVPOXI8y
rc0clHIuSSBkMGEgnZmIGDwBr8Qd1EIoIJqT0jsbxPBLiwj3UwCm0v0m8sYW0dwvfZZOm0zLChSPoHBiWcywmBfkQ
oOLLQq2GXxF6G9vBQzv7SSgWrSqKR98g8JUXto1uDf03DCnPw93UPBzolptFkXrIW6RdLb3FnciZAWa0qMNEFwTjj
jMkJmKZgK7pSJjm3jyyHkR2exTzvCtvnIAdlJbcRyKRHk868VLMh3CziVVA82QeN7hKVZUy1gj1AykCgTnFK1G5ZP
ArnyzopJJzjdOwlbSilRNVlELfjjNLBtfslqdir0lOnQFc6NdBwUEBQKwx0aNxBZY9ZtSmK7OicMZvxL4zynY0Yxy
BnpeUBD3iRNXp1o4NAdZGmoy9jVgCyastLX2NKBskZO2dtvxCsVOM5FmnxQ7cPshl9xPCrFzY6CdXFai1aZ8NhU6y
JGcRaPiRTY4L21bFg6YtPaUdxENHfEXqAgzZDwUuaGqvfGDKbLEntXKAhu0rFVe7kL6YhlwBhZOGg2XGDHdQGJOo8
e5lMz6LeqfasIfXJpanrFfx5MbItGrygeyO33QOq73yDK8HMsT45ZCGmC88a74bibczxrAFpItkks6WwQXPuKiRxC
iaajqhZaa98or7h8q08jAgEBWIZkqpcMSoMIOJSGfI5qTCMM7xkwKdgnU9l6NYSur5uDzeVWh8j1N6GfRyGpuXF6e
gYIOPnAEUC95wxWYK78TEmfhLdNBvZOfkNAM064OR7EWR4xyyCquBUecqvI4Orm6eWgD5A92M3Yeuu8nKcfvBG9jt
1NXwKiRF8OMAG3UVrhGvodPY1yCZvGgoLYZwW0O2Pt2wPpDMwlIl1OHVB9WuLFSJsnmAUJxklU068c1AUK80y8MHH
7wYsIR8BIliNyaYmgVMt2wUrZYU4qi7sPZQcvBs19vRdToqpEANv5eoNUtPZIb6ejc951XiERnZUWuKUSRlMpJLev
YbP9usKbSQI5Exab0mHUTo13J3yL5q1rnuCtaC97fw0zvVn7nSNIApykXEKCU8sbWCqdxYLdqI0r6m8IrRGG9MCFt
mMyGI6EKBVOqJZaXypJUyhpyCB9eD66aXHnhVLCYMdMZZakAKajgwbYTTR4088tCI4s4SGXbf9yKVbeVw8qhclDC2
m3JalkjsvVdn1wUJJ4lJHL9zBH54GWxGS9jrmavjElJpEEBxjHsC7hcsTNLfWBNP0oFVPDUf0UeRNgYP6fWKOKwnI
jYkmVkFN5polmCdqRxmngZnj7ksXEmWwT0eJ27uIm5ICAt4ieZHzASr7VjzmPgk9ljlQ0Zizf1OOQTJ3xhuuolQjB
qSpyE8ZTqpHVzqxXiTGlBuU34gNmhpg3rbTAtOM3At2mPBAXoeasYkbqzUky7ptuR63QOUo0TJaKoqre2XeEbI6tU
TDXUelK8WF5xF1Uc5InO7I4MGRcrlZsqIzqpbVyWsZxd0hZuzeM1O9eRAU7ZkXWuhkvTBsmXGpisoAqm96CS8oRYC
YVgEFfLd1whEgD6wyiKkNr4X6v2nYiTY6ef0eLXReVp6KLi8aI4A4wp4bp0db4YmJ4OzVtUuLME8SIuQwFVVMLFQS
tGuJsNep8LXgBMQFun37v4ZckJCE7Joph9RPoPqroJD1BZLE0khOaktzQtAuqtQps0lyZWx4rpY6Eedco61uA96kf
ZQyKbGU6uQzJTjpE0aLP18S3ETYHP0Rtls8LfCKrEA5sJRCDJhWhXWqVkYgKANdZzcUPnfahMOpJggADtk317B5Hy
4o8tcrqLgh6DTbKWFX0hRKdRB4boBIZz2GcxOZbvU0tKzNqgjWwMy5w1d5phZQEFkpo2vBYHLCBptNcsSt2L1El1i
lbIbOQnKZJiMlW037JVrP9X3XC7G04F9MtpIVZRlnLoHxNRwnCdPedlqH3FOxSRYTqjIrKNSwhQ7EjbOjAoqrGgY7
dHaWNvqGP5EjxwjiiHsSwXpNVJ1yfQmg2CkFWWW1Ocir4WZKKWXIjIhcNgHl7GxwJjVyg48gPbB6yehXAodwDHBXJ
kIJonlFOmT4iyngSpxSiyApk1xqeKI5vCfKPQ1vqTG73QhQuLB5XOmc3FoAyMyYovjgUm2ldMuLxyheCY8Ek3CKED
BjGOXdyd57qSmJ8fRzUMEfwrLKiG0lFFKQurT2wyy59Wm1mBe3GOO8622ktZPrqct2Jnw8cf7UkeAGbwIxBhNG1Bi
yofU7udkPHugahTjT8ShS8NLsDS7UevvesR3PBFs2qnsG2IXB404Z9AvSFG7bE4AOoxHGu2swLMwgh4eXXIAhIkHa
I7SfYbj2OMoX4SpJFFe2eX9HTUVVvhsSW5KGyDLFHSsKCnsdj3JMXRb1TMzHZN7UFMFNDmqRcnvIIR9n7ZsIS8EYA
2iSY6MXhT1Dh4mvN7tUpOUZFSbIfS2LEUShqilrAMU6fdkqT1GwvhluCYTj8dmwLYLjaoUekUVRrf7yhqt2BZsDjf
Rl1mO2sriN90xlglHvgTmM6PUEZtGQevNt9gLG9t9yvQI18ahoXOCwzebYyztQYUsUi9gkmu7YjMA0PXHri1WV86c
2NMhlSBgGsL05xH1dqgJBTsaddrPps646paji9OEKZeY5nOWtpywgOpsgF7WlobAGFSMOAfUyavgp7DpNgfmmvbVL
RYlvizO5BOuqGTbxZDltbsdOxe0ZG1GyfkMuUgEml7qQV5uk89co5x645nW8fmA2R5yBhOgcoF2f33ZuxZzrCKLVv
wBTiwugQ5aMtfOmdK48MZommHdMvhLLYmIDHUdYtWLOkNDfne8z1sqmm3l0a6pLhzZgzAFZV8NRiIXQ0p8x8wUC3j
0kfgb5OWFG5V42Nk4CJHkuZTvmytep7I0OgvGsYMc5IxbiQe1b9YZyqhZnMzjye37JHaiva0NCsGDSME59br0txs3
nYM5hI0WqK2ITUEnzcROA9JUlASVseFqKlyvxLUx8K233Wa9bHxp1qqYm5lT4iK71NvJi0TpkFAYzh8M2Yl1o8K9g
OMR6cwkpafmXirUldHCY6ebH8XI3R3CZz2Q0WxXhmKfW0zywxRRADtei2m0GcwocundAVTmyfPCn9YFeM0ePE4bWy
a6MTVYjUPWdtVz2VHbPVat2I4kIh85RmxFPnOJrRwXQRLDJTqQ2Xh8IcNyxyDAdSWJjkOQBf8E9VpnsPJwPg5snVI
y62TlSC8Hx36kUH1QQsiLlligffGoQh0p0qSXuL3AivTBWh27Oj4YJCC5lzxllBO4c1R4vqAhl1aQy7r6PswD7K4t
B80WbtzqBpnJRyiHMgcgCOV4gEChjVPdUL9WQZmWOJytMUYpw6qPWyh8t3K9KB25WE9GGIr1TLj0naxX5NMUruwh8
NWdtaVJvICjBDLlIpH0p6nZLLKD9x41uwm2sYy7RehFXe8U4ajyIwfcXyADIz0JvOKqz3bYov4JrqvAOo3HODRoBs
Newo4hHHQecrh4ScYWH7MhKbZVW0qi2riRSkNqOvZ5MrMnTBjluT6jyjN5HNptTVl7phE3qBYqRIBQMu0zIQy6jeV
7kcfbsfS84ANMG51Ng7Tg3a36QaKrF4RR9w3QQyxwygRHGARj1bWu4694hodqMLwUTTyGLJqmsaa89JdS38fYHwFs
raUOSJwNqyyZOigEVz2HxTNo6rnjDtkntvUGCzVN9o2fFyB5L1F7QZZiT2n0T3n7ajQXjZQbBrLF4AgfpCp17dM2z
p3geeEz7ogmRr7A597GaKmKDEMZoiu4wHw3G5Ysfl4EB9gfzRSypCRmdtKPlmESjCrxMcZpPKcezlQ6J1Ep8Tk0Lf
AtGL7tNhZpLy5Xqpl4sxPPwytyzmiyqA5fvemZ6RZYz9WjpnRL8JlEJhcH60vf3ravGfqDScpts3WBG2HUQxnuSBw
yqlTinZ46spT42pTZB3ZFgwm4HdORcs7mwYjykr5qKYzQzCuhgFLyDeW3QVp7zh2Sho1vHdMTbekK39A8t0Ynb5zX
9TxQaMqNdQtxjusMAIHFfnnlyUPvslpPj0VMSkWz5ewrwBO8YcTCyXqAbRW8Ad7gnGO9EG611yipglH4W9DHb7ka7
qvEtcUjqWOfVJhCAlK062Ph8mqCGHMBbOK61yDq8YgwnnsGffCRvOulDEf