A Word Sampler for Well-Typed Functions

Breandan Considine
bre@ndan.co

Abstract

We describe an exact sampler for a simply-typed, first-order functional programming language. Given an acyclic finite automaton, $\alpha_\varnothing$, it samples a random function uniformly without replacement from well-typed functions in $\mathcal{L}(\alpha_\varnothing)$. This is achieved via a fixed-parameter tractable reduction from a syntax-directed type system to a context-free grammar, preserving type soundness and completeness w.r.t. $\mathcal{L}(\alpha_\varnothing)$, while retaining the robust metatheory of formal languages.

1 Introduction

Consider a simply-typed language with the following terms:

```
FUN  ::=  fun f0 ( PRM ) : 𝕋 = EXP
PRM  ::=  PID : 𝕋 | PRM , PID : 𝕋
EXP  ::=  ⌜ℕ⌟ | ⌜𝔹⌟ | PID | INV | IFE | OPX
OPX  ::=  ( EXP OPR EXP )
IFE  ::=  if EXP { EXP } else { EXP }
INV  ::=  FID ( ARG )
ARG  ::=  EXP | ARG , EXP
OPR  ::=  + | * | < | ==
PID  ::=  p1 | ... | pk
FID  ::=  f0 | f1 | ... | fn
⌜𝔹⌟  ::=  true | false
⌜ℕ⌟  ::=  1 | 2 | 3 | ...
```

At the type level, we will assume an ambient global context, Γ, consisting of invokable named functions, and a finite type universe with two primitive types, $\mathbb{B}$ and $\mathbb{N}$.

$$
\begin{array}{rcl}
\Gamma & ::= & \varnothing \quad | \quad \Gamma, \texttt{f\_} : (\tau_1, \ldots, \tau_k) \to \tau \\
\mathbb{T} & ::= & \mathbb{B} \quad | \quad \mathbb{N} \quad | \quad \tau^{(3)} \quad | \quad \ldots \quad | \quad \tau^{(d)}
\end{array}
$$

Let us define a fragment of the typing judgements for IFE, INV, and OPX, which are mostly conventional.

$$
\frac{\Gamma \vdash e_c : \mathbb{B} \qquad \Gamma \vdash e_\top : \tau \qquad \Gamma \vdash e_\bot : \tau}{\Gamma \vdash \texttt{if}\ e_c\ \texttt{\{}\ e_\top\ \texttt{\}}\ \texttt{else}\ \texttt{\{}\ e_\bot\ \texttt{\}} : \tau}\ \texttt{IFE}
$$

$$
\frac{\Gamma \vdash \texttt{f\_} : (\tau_1, \ldots, \tau_m) \to \tau \qquad \Gamma \vdash e_i : \tau_i\ \forall i \in [1, m]}{\Gamma \vdash \texttt{f\_}\ \texttt{(}\ e_1\ \texttt{,}\ \ldots\ \texttt{,}\ e_m\ \texttt{)} : \tau}\ \texttt{INV}
$$

$$
\frac{\delta_{\texttt{OPR}}(\odot, \tau, \tau') = \hat{\tau} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash \texttt{(}\ e_1 \odot e_2\ \texttt{)} : \hat{\tau}}\ \texttt{OPX}
$$

where $\delta_{\texttt{OPR}} : \Sigma_{\texttt{OPR}} \times \mathbb{T} \times \mathbb{T} \rightharpoonup \mathbb{T}$ is defined as follows:

$$
\delta_{\texttt{OPR}}(\odot, \tau, \tau') = \begin{cases} \mathbb{B} & \text{if } \odot \in \{\texttt{<}\}, \tau, \tau' : \mathbb{N} \\ \mathbb{N} & \text{if } \odot \in \{\texttt{+}, \texttt{*}\}, \tau, \tau' : \mathbb{N} \\ \mathbb{B} & \text{if } \odot \in \{\texttt{==}\}, \tau = \tau'\ \ \forall \tau, \tau' : \mathbb{T} \end{cases}
$$

We will encode the type checker as a context-free grammar.

2 Notation

Recall that a context-free grammar (CFG) is a quadruple, $\langle \Sigma, V, P, S \rangle$, consisting of terminals (Σ), nonterminals (V), productions $\big(P \subset V \times (V \cup \Sigma)^*\big)$, and a start symbol ($S$). Also, a finite automaton (FA) is a quintuple $\langle Q, \Sigma, \delta, q_\alpha, F \rangle$, with states ($Q$), an alphabet ($\Sigma$), transitions ($\delta \subseteq Q \times \Sigma \times Q$), an initial state ($q_\alpha$), and accepting states ($F \subseteq Q$). These devices generate words in languages, denoted $\mathcal{L}(\cdot) \subseteq \Sigma^*$, that are context-free and regular, respectively.

A few notational rules for CFG compilation will be helpful:

$$
\frac{\texttt{.}}{\texttt{.} \in \Sigma}\ \Sigma \qquad \frac{(\sigma_0 \to \sigma_{1..n}) \in P}{\bigcup_{i=0}^{n} \{\sigma_i\} \setminus \Sigma \in V}\ P_V \qquad \frac{(\sigma_0 \to \sigma_{1..n}) \in P}{\bigcup_{i=1}^{n} \{\sigma_i\} \setminus V \in \Sigma}\ P_\Sigma
$$

The notation $\mathop{\boldsymbol{,}}(\cdot)$ is a macro for a comma-separated list.[1]

3 Method

We want to permit functions of up to arity-k, so the start symbol, S_Γ, will need to express each of these possibilities:

$$
\frac{\langle \vec{\tau}, \dot{\tau} \rangle \in \mathbb{T}^{0..k} \times \mathbb{T} \qquad \vec{\tau}_{0..|\vec{\tau}|} \in \vec{\tau}}{\Big(S_\Gamma \to \texttt{fun}\ \texttt{f0}\ \texttt{(}\ \mathop{\boldsymbol{,}}\limits_{i=1}^{|\vec{\tau}|} (p_i\ \texttt{:}\ \vec{\tau}_i)\ \texttt{)}\ \texttt{:}\ \dot{\tau}\ \texttt{=}\ \texttt{EXP}[\dot{\tau}, \vec{\tau} \to \dot{\tau}]\Big) \in P_\Gamma}\ \texttt{FUN}_\varphi
$$

We will decorate EXP nonterminals with a pair, $\texttt{EXP}[\cdot, \cdot]$, of (1) the expression's local return type (τ), and (2) available parameters ($\vec{\tau}$) and expected return type ($\dot{\tau}$) for $\texttt{f0} : \vec{\tau} \to \dot{\tau}$:

$$
\frac{\texttt{EXP}[\tau, \vec{\tau} \to \dot{\tau}] \in V_\Gamma \qquad \Gamma \vdash \texttt{f\_} : (\tau_1, \ldots, \tau_m) \to \tau}{\big(\texttt{EXP}[\tau, \vec{\tau} \to \dot{\tau}]\ \to\ \texttt{f\_}\ \texttt{(}\ \mathop{\boldsymbol{,}}\limits_{i=1}^{m} \texttt{EXP}[\tau_i, \vec{\tau} \to \dot{\tau}]\ \texttt{)}\big) \in P_\Gamma}\ \texttt{INV}_\varphi
$$

$$
\frac{\texttt{EXP}[\tau, \vec{\tau} \to \dot{\tau}] \in V_\Gamma \qquad \tau = \dot{\tau} \qquad \vec{\tau}_{0..|\vec{\tau}|} \in \vec{\tau}}{\big(\texttt{EXP}[\tau, \vec{\tau} \to \dot{\tau}]\ \to\ \texttt{f0}\ \texttt{(}\ \mathop{\boldsymbol{,}}\limits_{i=1}^{|\vec{\tau}|} \texttt{EXP}[\vec{\tau}_i, \vec{\tau} \to \dot{\tau}]\ \texttt{)}\big) \in P_\Gamma}\ \texttt{REC}_\varphi
$$

$$
\frac{\texttt{EXP}[\tau, \vec{\tau} \to \dot{\tau}] \in V_\Gamma \qquad \tau = \tau' \qquad \tau, \tau' \in \mathbb{T}}{\left(\begin{array}{c} \texttt{EXP}[\tau, \vec{\tau} \to \dot{\tau}]\ \to\ \texttt{if}\ \texttt{EXP}[\mathbb{B}, \vec{\tau} \to \dot{\tau}]\ \texttt{\{}\ \texttt{EXP}[\tau, \vec{\tau} \to \dot{\tau}]\ \texttt{\}} \\ \texttt{else}\ \texttt{\{}\ \texttt{EXP}[\tau', \vec{\tau} \to \dot{\tau}]\ \texttt{\}} \end{array}\right) \in P_\Gamma}\ \texttt{IFE}_\varphi
$$

$$
\frac{\texttt{EXP}[\hat{\tau}, \vec{\tau} \to \dot{\tau}] \in V_\Gamma \qquad \delta_{\texttt{OPR}}(\odot, \tau, \tau') = \hat{\tau} \qquad \odot \in \{\texttt{==}, \texttt{<}, \texttt{+}, \texttt{*}\}}{\big(\texttt{EXP}[\hat{\tau}, \vec{\tau} \to \dot{\tau}]\ \to\ \texttt{(}\ \texttt{EXP}[\tau, \vec{\tau} \to \dot{\tau}]\ \odot\ \texttt{EXP}[\tau', \vec{\tau} \to \dot{\tau}]\ \texttt{)}\big) \in P_\Gamma}\ \texttt{OPX}_\varphi
$$

$$
\frac{\texttt{EXP}[\tau, \vec{\tau} \to \dot{\tau}] \in V_\Gamma \quad \exists \vec{\tau}_i = \tau}{\big(\texttt{EXP}[\tau, \vec{\tau} \to \dot{\tau}] \to \texttt{pi}\big) \in P_\Gamma}\ \texttt{PID}_\varphi \qquad \frac{\texttt{EXP}[\tau, \vec{\tau} \to \dot{\tau}] \in V_\Gamma \quad \tau \in \{\mathbb{B}, \mathbb{N}\} \quad \texttt{\_} : \tau}{\big(\texttt{EXP}[\tau, \vec{\tau} \to \dot{\tau}] \to \texttt{\_}\big) \in P_\Gamma}\ \texttt{LIT}_\varphi
$$

The resulting grammar, $G_\Gamma := \langle \Sigma, V_\Gamma, P_\Gamma, S_\Gamma \rangle$, will be put into Chomsky Normal Form (CNF), G'_Γ, pruning productions containing unreachable or unproductive nonterminals and refactoring each production to either $(w \to xz) : V \times V^2$ or $(w \to t) : V \times \Sigma$. During normalization, arbitrary CFGs may undergo a quadratic blowup in space [6], however, as this grammar does not use ε or contain unary production chains, we can approximate the enlargement as being linear in $|G_\Gamma|$.

[1] i.e., $\mathop{\boldsymbol{,}}\limits_{i=1}^{m} (x_i)\ :=\ x_1\ \texttt{,}\ \ \ldots\ \ x_m$ if $m > 1$ else x_1 if $m = 1$ else ε.

3.1 Space complexity

Let us attempt to estimate $|G'_\Gamma|$ in terms of the contribution from each constructor. Following the convention of Lange and Leiß [6], we define $|G| = \sum_{w \in V} \sum_{w \to \sigma} |w\sigma|$. We will also use $|\cdot|'$ to denote binarized production size, which depends on the specific binarization technique, but is bounded by:

$$|w \to \sigma|' : P \to \mathbb{N} \begin{cases} = |w\sigma| & \text{if } |w\sigma| \leq 3 \\ \leq 3|w\sigma| - 1 & \text{otherwise.} \end{cases}$$

The leading term clearly depends on FUN_φ, which generates function signatures up to arity-k, with $d = |\mathbb{T}|$ types. If one considers permuted orderings of the input type signature, $\vec{\tau}$, as identical, its cost improves to $d \sum_{i=0}^{k} \binom{d+i-1}{i-1}$, however we will adhere to the naïve interpretation, which takes an arithmetico-geometric form, $\sum_{p=1}^{k} p d^{p+1}$, whose Lange-Leiß size is primarily determined by three factors:

$$|\mathsf{FUN}_\varphi|' = \Big|S_\Gamma \to \texttt{fun f0 (} \mathop{\boldsymbol{,}}_{i=1}^{|\vec{\tau}|} p_i \texttt{ : } \vec{\tau}_i \texttt{) : } \tau \texttt{ = } \mathsf{EXP}[\tau, \vec{\tau} \to \tau]\Big|' \leq \boxed{12|\vec{\tau}| + 23}$$

$$|\mathsf{REC}_\varphi|' = \Big|\mathsf{EXP}[\tau, \vec{\tau} \to \tau] \to \texttt{f0 (} \mathop{\boldsymbol{,}}_{i=1}^{|\vec{\tau}|} \mathsf{EXP}[\vec{\tau}_i, \vec{\tau} \to \tau] \texttt{)}\Big|' \leq \boxed{6|\vec{\tau}| + 8}$$

$$|\mathsf{PID}_\varphi|' = |\vec{\tau}| \cdot \Big|\mathsf{EXP}[\vec{\tau}_i, \vec{\tau} \to \tau] \to \texttt{pi}\Big|' = \boxed{2|\vec{\tau}|}$$

Letting $p = |\vec{\tau}|$ and assembling these factors, we have,

$$|G'_\Gamma| \simeq \sum_{p=0}^{k} d^{p+1} \Big(\underbrace{(12p + 23)}_{|\mathsf{FUN}_\varphi|'} + \underbrace{(6p + 8)}_{|\mathsf{REC}_\varphi|'} \Big) + \sum_{p=1}^{k} d^{p+1} \underbrace{(2p)}_{|\mathsf{PID}_\varphi|'}$$

$$\simeq \sum_{p=1}^{k} (20p + 31) d^{p+1} \simeq \frac{20kd^{k+2}}{(d-1)^2} + \mathcal{O}\left(\frac{d^{k+2}}{d-1}\right) + \ldots$$

and lower-degree terms. While our analysis omits INV_φ, et al., their contributions are less sensitive to the parameter k.

3.2 Sampling

The context-free languages have the pleasant property of being closed under intersection with regular languages [1], with an explicit construction given by Salomaa [9]. In brief, for every production $\mathrm{W} \to \mathrm{X\,Z}$ in the CNF grammar and state triple $p, q, r : Q$ in the automaton one creates synthetic (1) binary productions $p\mathrm{W}r \to p\mathrm{X}q\, q\mathrm{Z}r$, (2) unit productions $p\mathrm{W}q \to a$ for every $\mathrm{W} \to a$ and p, q such that $\delta(p, a) = q$, and (3) start productions $\mathrm{S} \to q_\alpha \mathrm{S} q_\omega$ for every final state q_ω.

Once constructed, a variety of methods for enumerating and sampling CFGs can be applied (e.g., [3, 4, 7]). We use Considine's [4], which supports sampling words in language intersections parameterized by an acyclic FA, in which case the intersection will be representable as an acyclic FA, $\alpha_\cap$.

This representation can be derminimized to produce an acyclic deterministic FA and then decoded left-to-right using an autoregressive model, or, if uniformity over $\ell_\cap$ is desired, by constructing a bijection, $b : \mathbb{Z}_{|\ell_\cap|} \leftrightarrow \ell_\cap$, and then drawing samples from a pseudorandom source (e.g., a linear feedback shift register). The latter method has the virtue of perfect parallelizability, although we evaluate this method serially.

4 Evaluation

We evaluate the sampler for small arity, $k \in [1, 3]$, with fixed $|\Gamma| = 18, |\mathbb{T}| = 7$ (see Appendix A). This generates tractable CNF grammars, $|G'_\Gamma| \in [1.9 \times 10^4, 9.9 \times 10^5]$, from which we then sample words on an Apple M4 with 16 GB of memory.

First, we sample words from a slice, $\sigma \leftsquigarrow \mathcal{L}(G'_\Gamma) \cap \Sigma^n$, and measure total time to first sample (μ TTFS). Next, using our dataset of random functions obtained during slice sampling, we will replace ($\texttt{:}\ \tau\ \texttt{=}$) with a hole ($\texttt{:}\ \Sigma\ \texttt{=}$) and resample $\sigma' \leftsquigarrow \mathcal{L}(G'_\Gamma) \cap (\ldots\ \texttt{:}\ \Sigma\ \texttt{=}\ \ldots)$, which we call type inference.

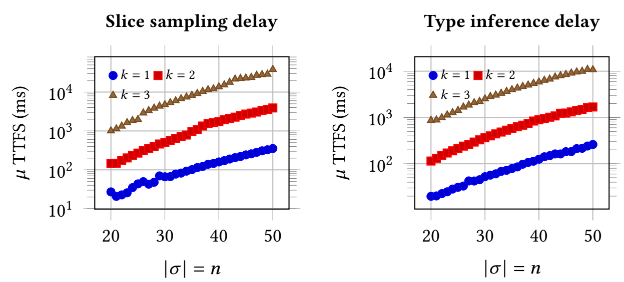

Figure 1. Slice sampling and type inference delay (log-scaled) vs. sequence length $|\sigma| = n$ for arities $k \in [1, 3]$.

Once the first sample is obtained, we observe bounded delay of 1786 ± 817 ns ($\mu \pm \sigma$), or an average throughput of $\sim 5.6 \times 10^5$ samples per second. Enumeration delay does not appear strongly correlated with either arity or word length.

5 Related work

Prior work demonstrates how to embed a deterministic CFL into a type-system [8], but the reverse direction remains largely unexplored. Existing work on constrained decoding (e.g., Willard et al. [10]) shows that syntactic soundness is feasible to guarantee, but the sample space is often ill-defined or an overapproximation to the space of semantically valid candidates. Frank et al. [5] introduce a type-theoretic method for sampling well-typed terms, but their method does not guarantee statistical uniformity or syntactic completeness. Finally, Bendkowski [2] uses techniques from enumerative combinatorics to sample closed λ-terms of the simply-typed variety, which is most closely related to this line of work.

6 Conclusion

We have presented a CFG embedding and exact sampler for well-typed functions, tractable for small-k. Extensions to straight-line programs, higher-order functions, and richer typing formalisms such as subtyping, parametric polymorphism, and substructural constraints are conceivable. Due to the cost of materializing G'_Γ, it would be advantageous to construct the constituent productions lazily, as only a small fraction may participate in a given language intersection. Another direction would be to collapse syntactic symmetries by quotienting productions, e.g., by semantic invariants or α-equivalence. We leave these possibilities for future work.

7 Acknowledgements

This paper is dedicated to Ori Roth, whose work on typelevel parsing inspired the author to pursue this direction.

References

[1] Yehoshua Bar-Hillel, Micha Perles, and Eli Shamir. 1961. On formal properties of simple phrase structure grammars. *Sprachtypologie und Universalienforschung* 14 (1961), 143–172.

[2] Maciej Bendkowski, Katarzyna Grygiel, and Paul Tarau. 2016. Boltzmann samplers for closed simply-typed lambda terms. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 120–135.

[3] David Chiang. 2007. Hierarchical phrase-based translation. *Computational Linguistics* 33, 2 (2007), 201–228.

[4] Breandan Considine. 2024. A Tree Sampler for Bounded Context-Free Languages. *arXiv preprint arXiv:2408.01849* (2024).

[5] Justin Frank, Benjamin Quiring, and Leonidas Lampropoulos. 2024. Generating Well-Typed Terms That Are Not "Useless". *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 2318–2339.

[6] Martin Lange and Hans Leiß. 2009. To CNF or not to CNF? An efficient yet presentable version of the CYK algorithm. *Informatica Didactica* 8, 2009 (2009), 1–21.

[7] Steven T. Piantadosi. 2023. How to enumerate trees from a context-free grammar. arXiv:2305.00522 [cs.CL]

[8] Ori Roth. 2021. Study of the Subtyping Machine of Nominal Subtyping with Variance. *arXiv preprint arXiv:2109.03950* (2021). https://arxiv.org/pdf/2109.03950.pdf

[9] Arto Salomaa. 1973. *Formal languages*. Academic Press, New York. 59–61 pages.

[10] Brandon T Willard and Rémi Louf. 2023. Efficient guided generation for LLMs. *arXiv preprint arXiv:2307.09702* (2023).

A Ambient context (Γ)

$$
\begin{aligned}
\texttt{i2s} &: \texttt{Int} \rightarrow \texttt{Str},\\
\texttt{s2i} &: \texttt{Str} \rightarrow \texttt{Int},\\
\texttt{i2f} &: \texttt{Int} \rightarrow \texttt{Float},\\
\texttt{len} &: \texttt{Str} \rightarrow \texttt{Int},\\
\texttt{concat} &: \texttt{Str} \times \texttt{Str} \rightarrow \texttt{Str},\\
\texttt{eqStr} &: \texttt{Str} \times \texttt{Str} \rightarrow \texttt{Bool},\\
\texttt{mkPair} &: \texttt{Int} \times \texttt{Int} \rightarrow \texttt{Pair},\\
\texttt{fst} &: \texttt{Pair} \rightarrow \texttt{Int},\\
\texttt{snd} &: \texttt{Pair} \rightarrow \texttt{Int},\\
\texttt{read} &: \texttt{Path} \rightarrow \texttt{Str},\\
\texttt{join} &: \texttt{Path} \times \texttt{Str} \rightarrow \texttt{Path},\\
\texttt{tmp} &: \texttt{Int} \rightarrow \texttt{Path},\\
\texttt{parseDate} &: \texttt{Str} \rightarrow \texttt{Date},\\
\texttt{dateToInt} &: \texttt{Date} \rightarrow \texttt{Int},\\
\texttt{addDays} &: \texttt{Date} \times \texttt{Int} \rightarrow \texttt{Date},\\
\texttt{isWeekend} &: \texttt{Date} \times \texttt{Bool},\\
\texttt{choose} &: \texttt{Bool} \times \texttt{Int} \times \texttt{Int} \rightarrow \texttt{Int},\\
\texttt{mux} &: \texttt{Bool} \times \texttt{Str} \times \texttt{Str} \rightarrow \texttt{Str}
\end{aligned}
$$

B Samples ($\sigma \leftsquigarrow \mathcal{L}(G'_\Gamma) \cap \Sigma^{28}$)

```
fun f0 ( p1 : Pair ) : Float = i2f ( choose ( ( 1 == 1 ) , snd ( p1 ) , 1 ) )
fun f0 ( p1 : Bool , p2 : Date ) : Str = f0 ( ( ( p1 == p1 ) == true ) , p2 )
fun f0 ( p1 : Bool , p2 : Bool ) : Path = f0 ( if p2 { p1 } else { true } , p1 )
fun f0 ( p1 : Bool ) : Pair = f0 ( ( ( 1 * len ( i2s ( 1 ) ) ) < 1 ) )
fun f0 ( p1 : Int ) : Int = choose ( false , 1 , ( f0 ( 1 ) * f0 ( p1 ) ) )
fun f0 ( p1 : Int ) : Bool = ( s2i ( i2s ( choose ( true , p1 , 1 ) ) ) < p1 )
fun f0 ( ) : Bool = if true { true } else { ( 1 < fst ( mkPair ( 1 , 1 ) ) ) }
fun f0 ( ) : Bool = ( false == ( ( 1 == ( 1 + 1 ) ) == ( 1 < 1 ) ) )
fun f0 ( p1 : Str , p2 : Bool ) : Int = if eqStr ( p1 , p1 ) { 1 } else { 1 }
fun f0 ( p1 : Int , p2 : Int ) : Pair = f0 ( ( p2 * ( p2 + p2 ) ) , p1 )
fun f0 ( p1 : Int ) : Float = i2f ( ( p1 + len ( i2s ( ( 1 * p1 ) ) ) ) )
fun f0 ( p1 : Bool , p2 : Int ) : Str = f0 ( ( ( 1 == p2 ) == true ) , 1 )
fun f0 ( ) : Int = if if true { true } else { false } { 1 } else { ( 1 * 1 ) }
fun f0 ( p1 : Bool , p2 : Bool ) : Int = ( f0 ( false , ( p1 == false ) ) + 1 )
fun f0 ( p1 : Str ) : Int = ( 1 + s2i ( i2s ( ( f0 ( p1 ) * 1 ) ) ) )
fun f0 ( p1 : Bool , p2 : Pair ) : Bool = ( f0 ( p1 , p2 ) == ( p1 == false ) )
fun f0 ( p1 : Bool ) : Bool = ( false == ( f0 ( p1 ) == f0 ( f0 ( p1 ) ) ) )
fun f0 ( ) : Bool = ( true == ( i2s ( 1 ) == i2s ( s2i ( i2s ( 1 ) ) ) ) )
fun f0 ( ) : Int = if false { ( 1 * 1 ) } else { ( 1 * ( 1 + 1 ) ) }
fun f0 ( ) : Int = ( choose ( false , 1 , f0 ( ) ) + choose ( true , 1 , 1 ) )
fun f0 ( p1 : Bool , p2 : Bool ) : Int = f0 ( p2 , ( ( 1 + 1 ) == 1 ) )
fun f0 ( p1 : Str ) : Int = f0 ( i2s ( ( ( 1 + f0 ( p1 ) ) * 1 ) ) )
fun f0 ( ) : Bool = ( ( true == ( false == ( ( 1 == 1 ) == false ) ) ) == true )
fun f0 ( ) : Bool = ( if true { 1 } else { ( 1 + 1 ) } == ( 1 * 1 ) )
fun f0 ( ) : Int = ( ( ( 1 + f0 ( ) ) + s2i ( i2s ( 1 ) ) ) * 1 )
fun f0 ( p1 : Bool , p2 : Bool ) : Int = f0 ( ( true == ( p1 == p2 ) ) , p1 )
fun f0 ( p1 : Int , p2 : Int ) : Bool = f0 ( ( 1 + p1 ) , ( p2 + p2 ) )
fun f0 ( p1 : Path ) : Int = choose ( ( p1 == p1 ) , 1 , len ( i2s ( 1 ) ) )
fun f0 ( p1 : Float , p2 : Pair ) : Bool = f0 ( p1 , mkPair ( snd ( p2 ) , 1 ) )
fun f0 ( p1 : Int , p2 : Int ) : Path = f0 ( ( p1 + ( p1 + p1 ) ) , 1 )
fun f0 ( p1 : Pair , p2 : Path , p3 : Str ) : Int = s2i ( i2s ( s2i ( p3 ) ) )
fun f0 ( p1 : Int ) : Int = ( ( len ( i2s ( f0 ( 1 ) ) ) * 1 ) + 1 )
fun f0 ( ) : Bool = ( ( false == ( true == ( false == true ) ) ) == ( 1 == 1 ) )
fun f0 ( p1 : Pair ) : Int = choose ( ( 1 < s2i ( i2s ( 1 ) ) ) , 1 , 1 )
fun f0 ( p1 : Str , p2 : Str ) : Bool = ( ( false == true ) == f0 ( p2 , p1 ) )
fun f0 ( p1 : Pair , p2 : Path ) : Pair = f0 ( p1 , join ( p2 , i2s ( 1 ) ) )
fun f0 ( p1 : Int ) : Pair = mkPair ( p1 , ( ( 1 + p1 ) * ( p1 * p1 ) ) )
fun f0 ( p1 : Pair ) : Pair = mkPair ( 1 , ( 1 + ( 1 + ( 1 + 1 ) ) ) )
fun f0 ( p1 : Int ) : Int = ( f0 ( f0 ( 1 ) ) + ( f0 ( p1 ) + 1 ) )
fun f0 ( ) : Bool = ( ( true == ( 1 == 1 ) ) == ( ( 1 + 1 ) == 1 ) )
fun f0 ( p1 : Pair , p2 : Pair ) : Int = ( 1 * ( f0 ( p2 , p2 ) + 1 ) )
fun f0 ( p1 : Int , p2 : Str ) : Pair = mkPair ( ( ( p1 + p1 ) + 1 ) , p1 )
fun f0 ( p1 : Str ) : Path = tmp ( ( s2i ( i2s ( ( 1 + 1 ) ) ) + 1 ) )
fun f0 ( p1 : Int ) : Int = ( choose ( false , f0 ( 1 ) , p1 ) + f0 ( 1 ) )
fun f0 ( p1 : Date , p2 : Int ) : Int = f0 ( p1 , ( p2 + ( p2 * p2 ) ) )
fun f0 ( p1 : Int , p2 : Bool ) : Pair = mkPair ( ( p1 * ( p1 + p1 ) ) , p1 )
fun f0 ( p1 : Bool , p2 : Bool ) : Date = f0 ( ( true == p1 ) , ( p1 == true ) )
fun f0 ( p1 : Str ) : Path = tmp ( ( 1 * choose ( true , s2i ( p1 ) , 1 ) ) )
fun f0 ( p1 : Int , p2 : Int ) : Float = f0 ( ( ( p2 + p2 ) * p1 ) , p1 )
fun f0 ( p1 : Bool ) : Str = concat ( f0 ( true ) , i2s ( s2i ( i2s ( 1 ) ) ) )
fun f0 ( p1 : Int , p2 : Path ) : Str = i2s ( ( p1 + s2i ( read ( p2 ) ) ) )
fun f0 ( p1 : Date , p2 : Float ) : Bool = ( len ( read ( tmp ( 1 ) ) ) == 1 )
```