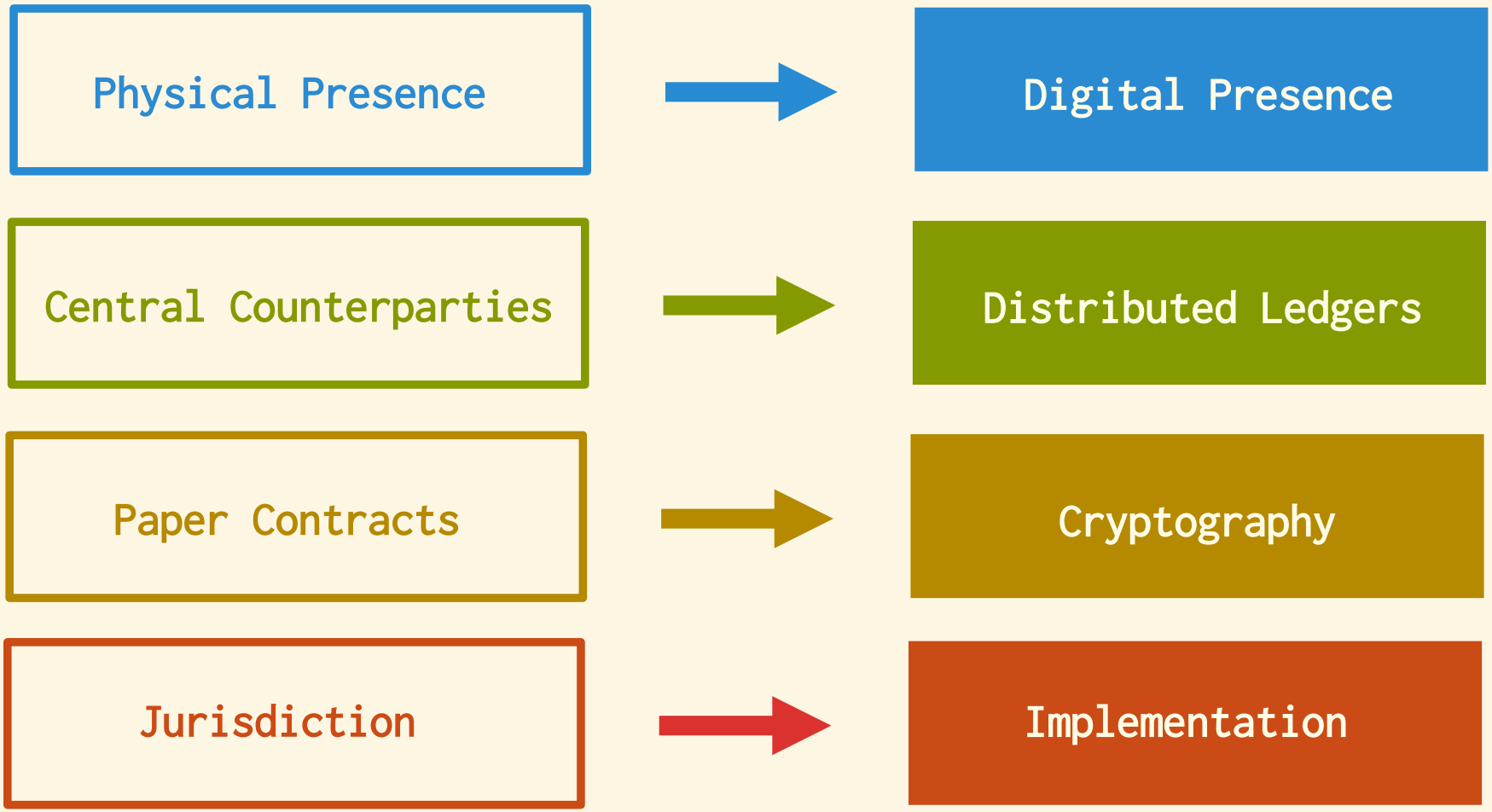
Yatima

A language for the dencentralized web

john@yatima.io

Part I: Motivation

Software Eats the Economy



This trend is just beginning

Blockchain = $10^11, Real Estate = $10^15

Problem: Software Breaks

- Leftpad: 2 hours global JS downtime
- Heartbleed: >$500 million
- The DAO: >$50 million
- Parity Multisig: >$100 million
- Knight Capital: >$500 million
- Meltdown/Spectre, 737-Max, Flash Crash, Mt Gox, etc. etc. etc...

Software Failures are unnatural disasters

Cost to US economy is trillions/year

Solution: Strong Types

Software	Mathematics
Type signature: `apply<P,Q>(f: P -> Q, x: P): Q {..}`	Theorem: $P \to Q,\ P \vdash Q$
Program: `|f,x| f(x)`	Proof: 
Type-checking programs	Verifying proof correctness
Specification: “Our requirements are ...”	Abstract: “In this paper we will prove ...”

Code **is** Math (modulo syntax)

Problem:
People Don’t Like Math

Solution: Syntax Genricity

```
// Procedural style like C, C++, JavaScript, Rust
apply<P,Q>(f: P -> Q, x: P): Q {
  f(x)
}
```

```
;;; S-expression style like  Lisp
(declare-type apply ((P Type) (Q Type) (f (fun P Q)) (x P)) Q)
(defun apply (P Q f x) (f x))
```

```
-- Equational-style like Haskell, Idris, OCaml, Lean
apply : (P: Type) -> (Q: Type) -> (P -> Q) -> P -> Q
apply f x = f x
```

```
// Yatima-Core syntax
def apply (P: Type) (Q: Type) (f: ∀ P -> Q) (x: P): Q
  := f x

def apply: ∀ (P: Type) (Q: Type) (f: ∀ P -> Q) (x: P) -> Q
  := λ f x => x
```

Solution: Syntax Genricity

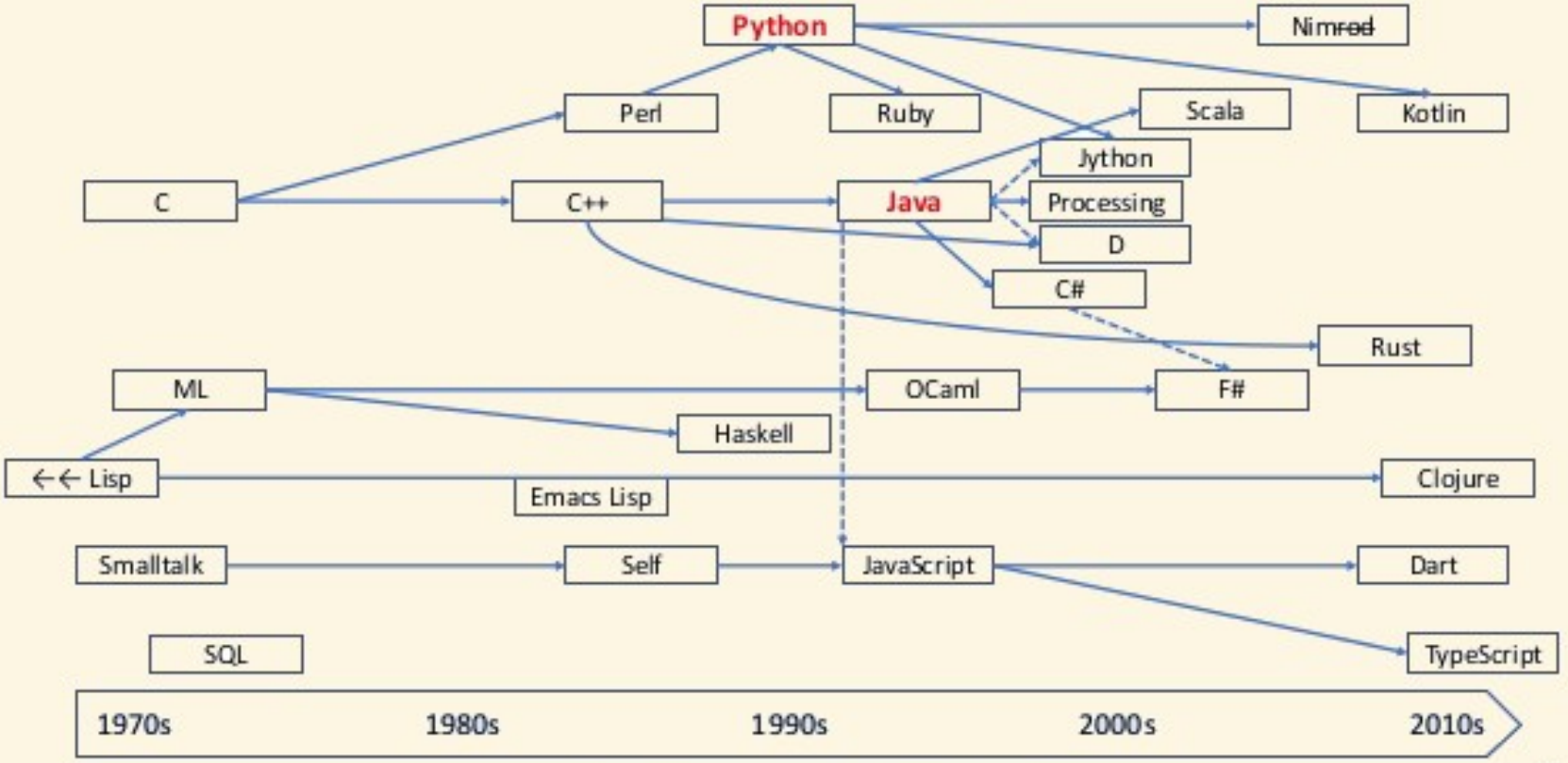
```
// Procedural style like C, C++, JavaScript, Rust
type List<A> {
  nil,
  cons(x: A, xs: List<A>),
}
```

```
;;; S-expression style like Lisp
(deftype List (A)
  (nil)
  (cons (x A) (xs (List A)))
)
```

```
-- Equational-style like Haskell, Idris, OCaml, Lean
data List A
  | Nil
  | Cons (x: A) (xs: List A)

data List A where
  Nil : List A
  Cons (x: A) (xs: List A) : List A
```

A Brief Family Tree of Some Programming Languages



Problem: People Don’t Like Code Either

Try to Solve Pain Points

- Focus on determinism over raw performance
- Same language everywhere (web, metal, ledger)
- Garbage-collection, but opt-out (linear types)
- Great languages have great communities (e.g. Rust)
- Build tools need to be safer and less arcane
- Docs need to be compiler-checked, web-rendered, type-indexed, searcheable, etc.
- Editor integration (language needs to run **in** editor)
- Different frontend syntaxes need to be interoperable

Part II: Implementation

Yatima Core

```
// 14 core constructors
pub enum Term {
  Var(String, u64),          // local variables (de bruijn indexed)
  Lam(String, Box<Term>),    // lambda abstraction
  App(Box<(Term,Term)>),     // application
  Ref(String, Link, Link), // global content-addressed reference
  Lit(Literal),              // data literal, like Text, Char, Integer
  Lty(LitType),              // type of Literals
  Opr(PrimOp),               // primitive operations over Literals.
  Let(bool, Uses, String, Box<(Term,Term,Term)>), // local definition
  All(Uses, String, Box<(Term,Term)>), // dependent function type
  Slf(String, Box<Term>),    // self-type type declaration
  Dat(Box<Term>),            // self-type constructor (datatype builder)
  Cse(Box<Term>),            // self-type eliminator (case expression)
  Typ,                       // the type of types is the type “Type”
  Ann(Box<(Term,Term)>),     // type annotation
}
```

Yatima Core

```
// 14 core constructors
pub enum Term {
  Var(String, u64),          // local variables (de bruijn indexed)
  Lam(String, Box<Term>),    // lambda abstraction
  App(Box<(Term,Term)>),     // application
  Ref(String, Link, Link), // global content-addressed reference
  Lit(Literal),              // data literal, like Text, Char, Integer
  Lty(LitType),              // type of Literals
  Opr(PrimOp),               // primitive operations over Literals.
  Let(bool, Uses, String, Box<(Term,Term,Term)>), // local definition
  All(Uses, String, Box<(Term,Term)>), // dependent function type
  Slf(String, Box<Term>),    // self-type type declaration
  Dat(Box<Term>),            // self-type constructor (datatype builder)
  Cse(Box<Term>),            // self-type eliminator (case expression)
  Typ,                       // the type of types is the type “Type”
  Ann(Box<(Term,Term)>),     // type annotation
}
```

λ-Calculus + Content Addressing + Primitives + Dependent/Self/Substructural Type System

λ-Calculus

- Invented in 1930s by Alonzo Church
- Universal model of computing

  (Church-Turing Thesis)
- Foundation of functional programming

Lam Var

(λf.λx.(f x))

App

Content-Addressing

- Store data by content rather than location
- Often using cryptographic hash functions
- e.g. Git, IPFS, Plan 9, Nix, Tahoe-LAFS, Unison
- Allows for peer-to-peer storage/lookup via a distributed hash table (e.g. Kademlia)
- Hashes are **proofs** of data immutability.
- “Code is Data”: Provably reproducible computation

```
⅄ #IVdLcYHSv5ipV8zpyTz28P2F8FpMLAGt5rncAXs6AxHcgvpsZH
→ (package MyPackage (imports (...)) (defs (...)))
```

Primitives

- Common datatypes like Text, Char, Integer, etc.
- Efficient low-level operations over those types
- Lazy expansion/compression of Literals to/from inductive datatypes.
- Dependent typing allows safe overloading
- Statically prevents overlow, div-by-zero
- Extensible, since they add no expressive power
- Exposed to user in addition to pure types and functions (unlike the “jets” approach used by Simplicity and Urbit)

```
人 #cat “foo” “bar” :: #Text
→ “foobar”
```

Dependent Types

- Function return type can change depending on input
- Allows for expressive “proofs-as-programs”
- Used in languages like Idris, Agda, Coq, Lean
- Can use term language in types, at the cost of undecidable checking
- Historically complex to implement in a practical programming language

```
ㅅ :type List.head
→ ∀ (A: Type) (x: List A) -> Subset (List A) List.NotEmpty
```

Substructural Types

- Functions can statically restrict runtime usage of arguments
- Used in Idris, Haskell, Rust, Clean
- Allows for safe resource usage, like files, locks, memory (like Rust borrow checker).
- Historically difficult to integrate with dependent types (Quantitative Type Theory)
- Yatima has a {0,&,1,ω} QTT usage multiplicity rig

```
⅄ :type (λ x => (x,x) :: ∀ (1 x: Int) -> Pair Int Int)
→ Error: Can't use linear variable 'x' twice
```

Self-Types

- Based on paper “Self Types for Dependently Typed Lambda Encodings” (Fu & Stump, 2014)
- Extends dependent types so that an expression’s type can depend on its value: `@self x :: P self`
- Used in the Cedille programming language and prover
- Allows construction of inductive datatypes
- Alternative models such as the Calculus of Inductive Constructions are much more complex

Datatype Encodings

```
// A datatype is encoded as its elimination function
// Identical to `data Bool = True | False`
def Bool : Type :=
  @self ∀
  (0 P    : ∀ Bool -> Type)
  (& true : P (data λ P t f => t))
  (& false: P (data λ P t f => f))
  -> P self

// Constructors
def true : Bool := data λ P t f => t
def false: Bool := data λ P t f => f

// Example: boolean negation
def not (a: Bool): Bool :=
 (case a)
   (λ self => Bool) // P
   false            // P self == P true == Bool
   true             // P self == P false  == Bool
```

Valus: A λ-graph VM

- (λ-Calculus + Content Addressing + Primitives + Dependent/Self/Substructural Types) interpreter
- Based on paper “Bottom-up β-reduction: uplinks and λ-DAGs” (Shivers & Wand, 2010)
- Fully lazy, efficient read-back, sharing
- Rust implementation compiles to x86 & WASM
- Yatima’s typechecker, can also act as runtime
  - (Or get x86/WASM binary via Chez Scheme/Schism)
- Better time/space computational cost metrics

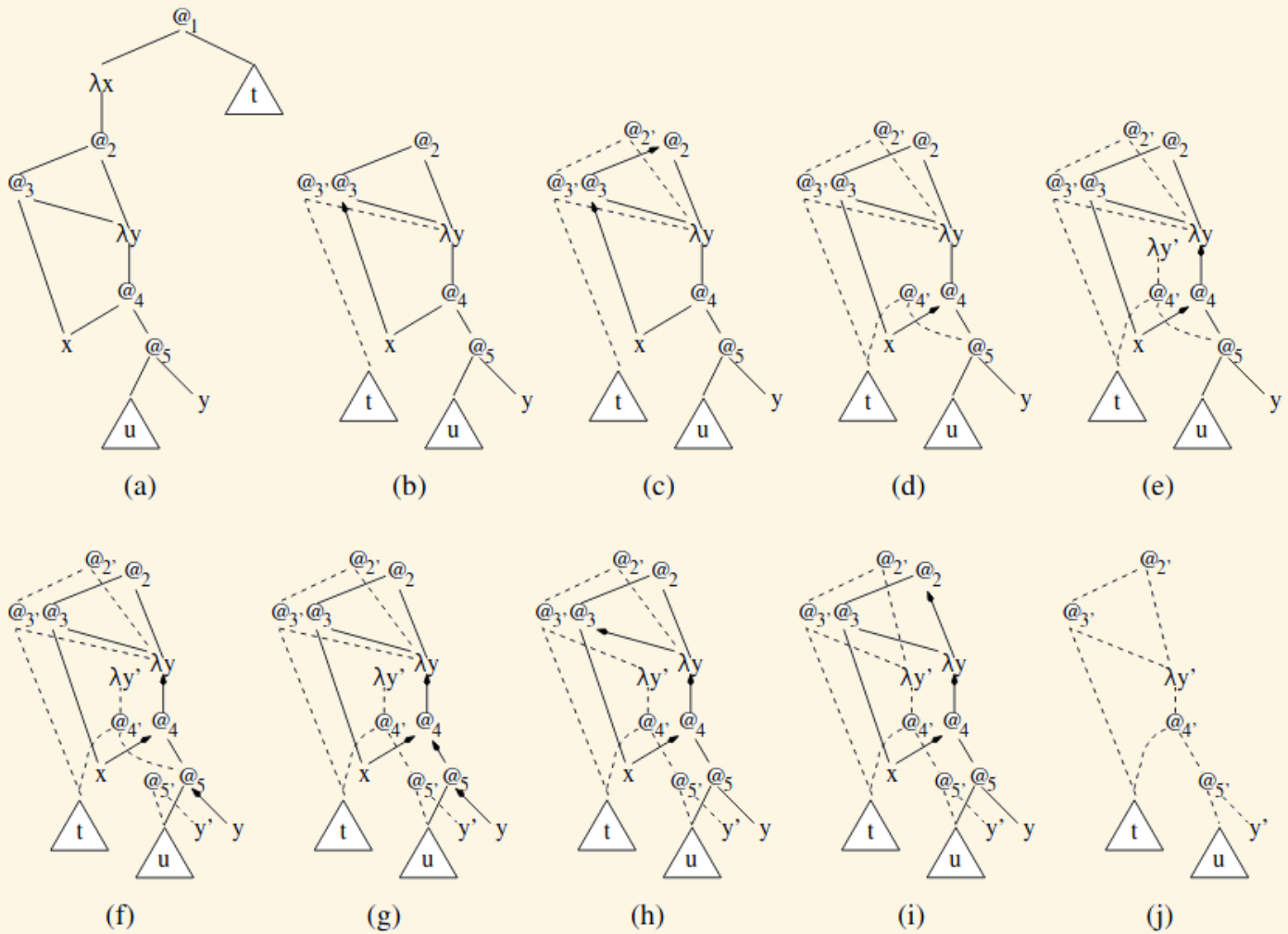


Figure 5: A trace of a bottom-up reduction of term $(\lambda x.(x(\lambda y.x(uy)))(\lambda y.x(uy)))t$, where the $(\lambda y.x(uy))$ term is shared, and sub-terms t and u are not specified.

"Bottom-up β-reduction: uplinks and λ-DAGs" (Shivers & Wand, 2010)

Hash-Expressions

- A new serialization format base on Rivest's Canonical S-Expressions (csexpr → hashexpr)
- Replaces IPFS's Multiformats/IPLD/CBOR/JSON stack
- Supports content-addressing of binary blobs, but is extensible to other types of serial data
- Extremely byte-efficient (based on MessagePack)
- Simple text format, with parsing and printing
- Encodes Yatima's packages and Abstract Syntax Tree
- Will integrate with Rust's Serde library in future

Hash-Expression encoding

```
// Yatima Definition, core syntax
def apply: ∀ (P: Type) (Q: Type) (f: ∀ P -> Q) (x: P) -> Q
  := λ f x => x
```

```
// Yatima Definition, hash-expression
(def apply ""
  (forall 0 P Type (forall 0 Q Type (forall ω f (forall ω P Q (forall ω x P Q))))
  (lambda P (lambda Q (lambda f (lambda x (f x)))))
)
```

```
// “desaturated” definition, separating computationally relevant and irrelevant data
(def apply ""
  #IvdLcYPuGB2yFswYBwSspoZswjwgYE8tYDnAJUJiGSWn542PPj // link to type anonymous AST
  #IvdLcYUZMwj3FVCuzfzNjGG4gprus7faJ7S8y8rycst3gsHiju // link to term anonymous AST
  // type name-metadata
  (ctor leaf () (bind P (ctor leaf () (bind Q
    (ctor leaf (ctor leaf leaf leaf) (bind f
    (ctor leaf leaf (bind x leaf)))
  ))))
  // term name-metadata of `λ f x => f x`
  (ctor (bind P (ctor (bind Q (ctor (bind f (ctor (bind x
    (ctor (ctor leaf) (ctor leaf))
  ))))))))
)
// anonymous AST of `λ f x => f x`
// #IvdLcYUZMwj3FVCuzfzNjGG4gprus7faJ7S8y8rycst3gsHiju
(ctor lam (bind (ctor lam (bind (ctor lam (bind (ctor lam (bind
  (ctor app (ctor var 1) (ctor var 0))
))))))))
```

Hashspace

- A data store for hash-expressions
- The backend for Yatima's package management
  - Reproducible, Reliable, Atomic builds
- Local cache in filesystem like NixOS /store/
- HTTP API for put/get on remote server
- Will add DHT for p2p-sharing like libp2p/IPFS
- Future features:
  - cloning of package sources
  - AST-aware merging/diffing

Part III: Integration (Discussion, Q&A)