Gemini

Designing and Implementing a Functional Hardware Description Language

Thesis Defense

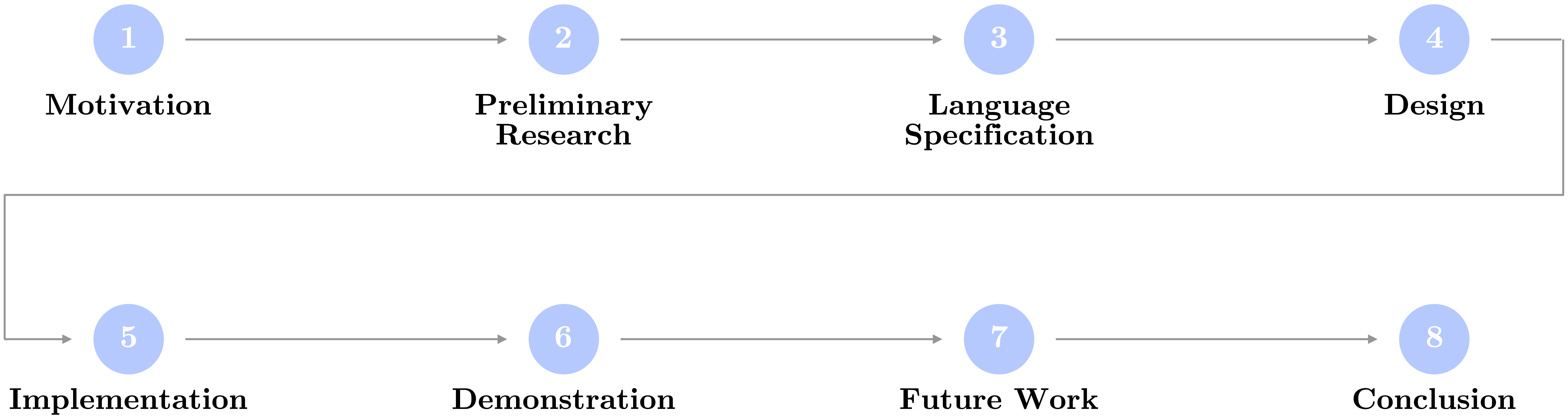
EDMUND T. PRATT, JR.
SCHOOL OF ENGINEERING

Aditya SRINIVASAN

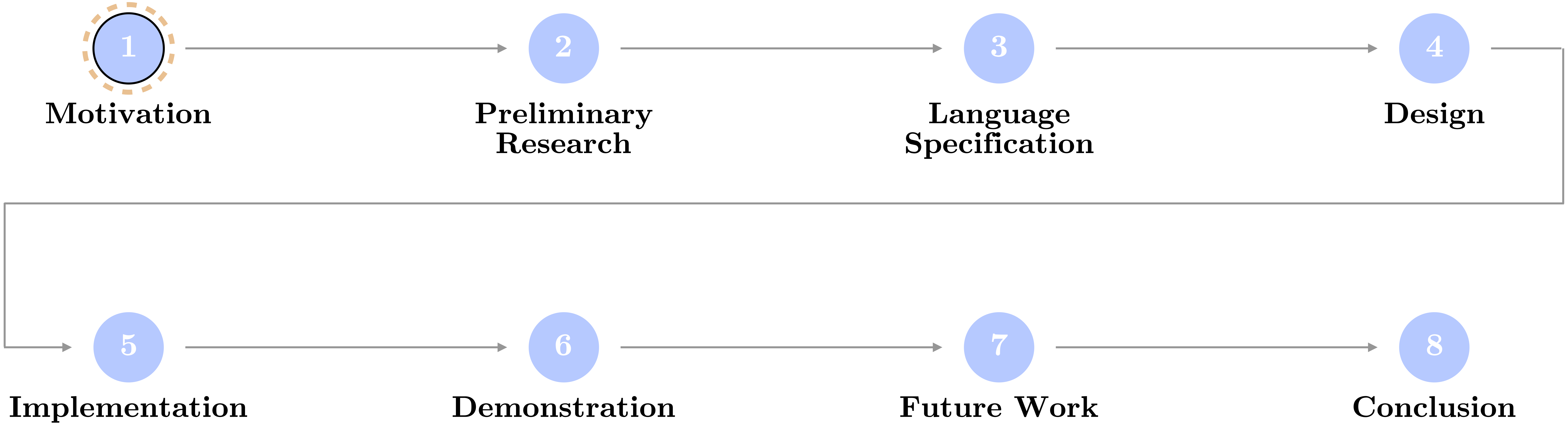
Drew HILTON

APRIL 25, 2018

Roadmap


1
Motivation
2
Preliminary Research
3
Language Specification
4
Design
5
Implementation
6
Demonstration
7
Future Work
8
Conclusion

Roadmap


1
Motivation
2
Preliminary Research
3
Language Specification
4
Design
5
Implementation
6
Demonstration
7
Future Work
8
Conclusion

Motivation

The Problem: Verilog lacks the expressivity and modularity of software programming languages, due to a lack of features such as:

- Strong type system
- Recursion
- Pattern-matching
- …and more

Motivation

I spent the last year answering the following questions:

Motivation

I spent the last year answering the following questions:

Question 1
Can I design a programming language that combines the powerful features of software programming languages with the ability to describe electronic circuits?

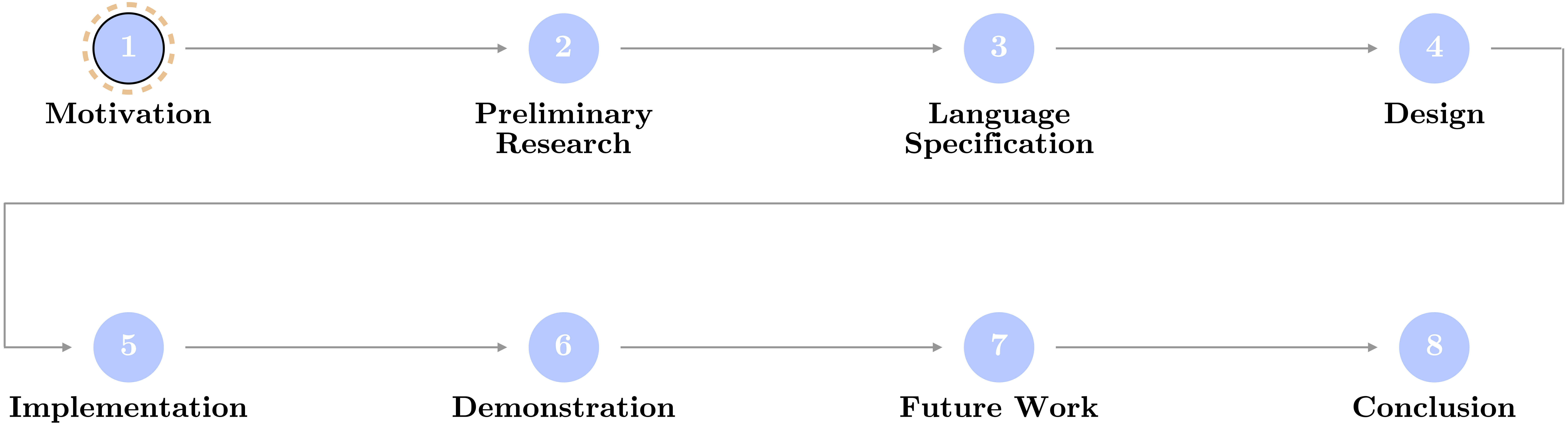
Motivation

I spent the last year answering the following questions:

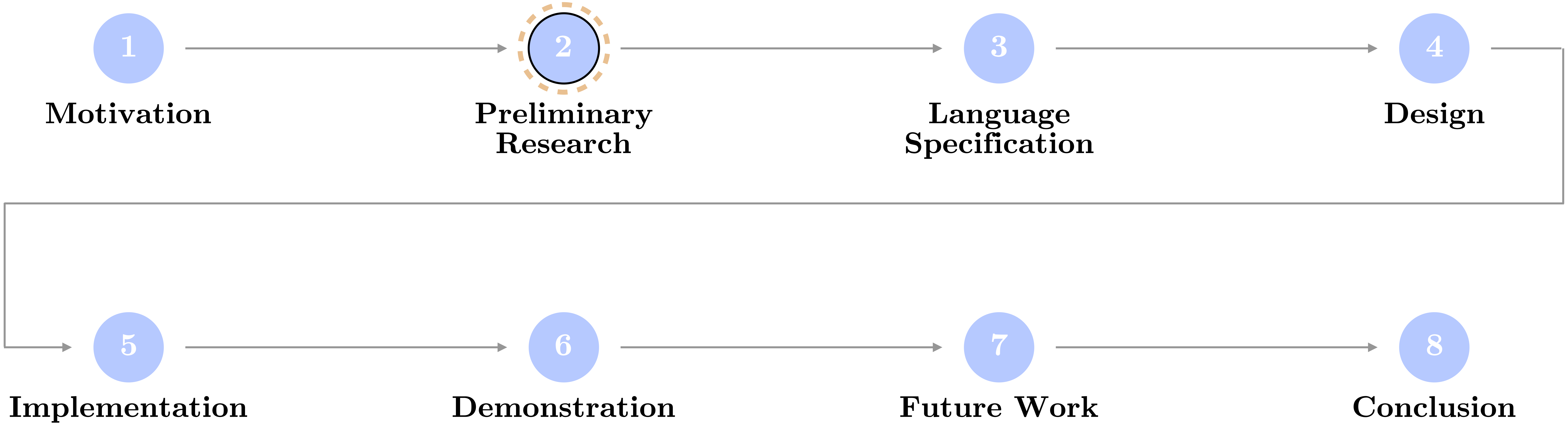
Question 1
Can I design a programming language that combines the powerful features of software programming languages with the ability to describe electronic circuits?

Question 2
Can I develop a compiler that accepts a program in this language and produces an optimized Verilog module?

Roadmap


1
Motivation
2
Preliminary Research
3
Language Specification
4
Design
5
Implementation
6
Demonstration
7
Future Work
8
Conclusion

Roadmap


1
Motivation
2
Preliminary Research
3
Language Specification
4
Design
5
Implementation
6
Demonstration
7
Future Work
8
Conclusion

Preliminary Research

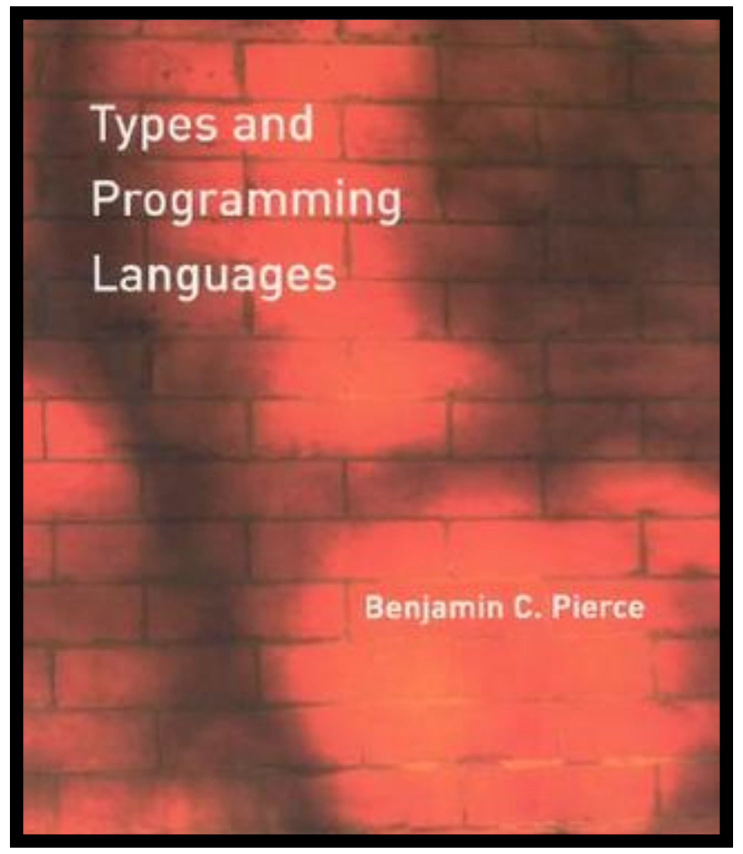
Possessed some prior knowledge through coursework

- ECE 350 (Digital Systems)
- ECE 553 (Compiler Construction)

Did not know enough about type theory to design a powerful language

Preliminary Research

Read the entirety of '<u>Types and Programming Languages</u>' by Benjamin Pierce, a textbook used in graduate-level type-theory seminars



Provided me with the theoretical tools I needed

Preliminary Research

Other languages exist that attempt to do the same thing

Haskell for Hardware (only software programmers can understand)

Lava (not high-level enough)

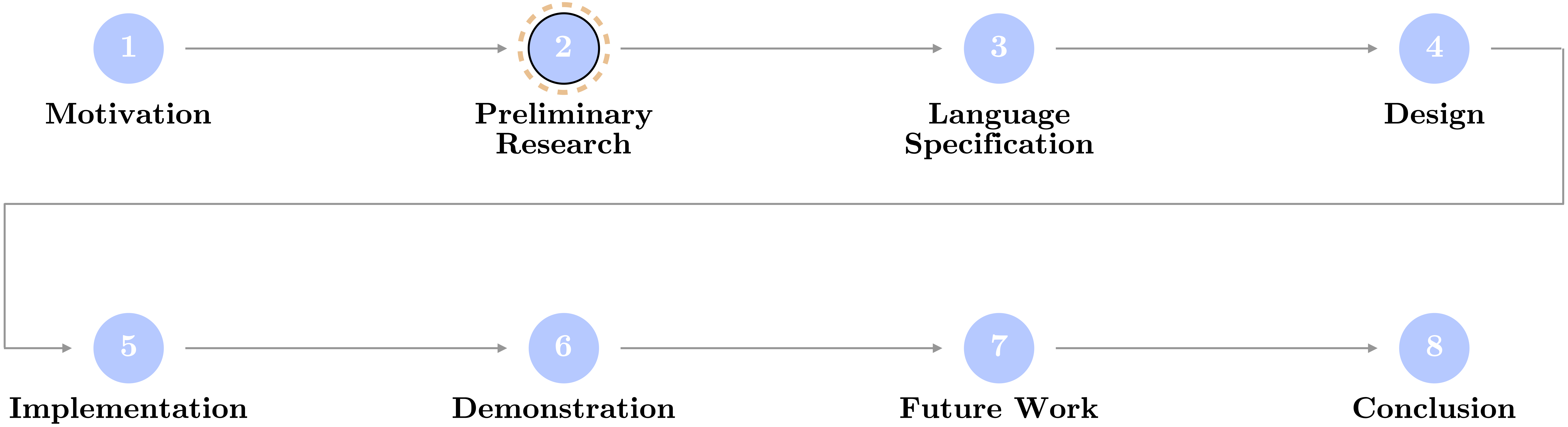
Preliminary Research

Other languages exist that attempt to do the same thing

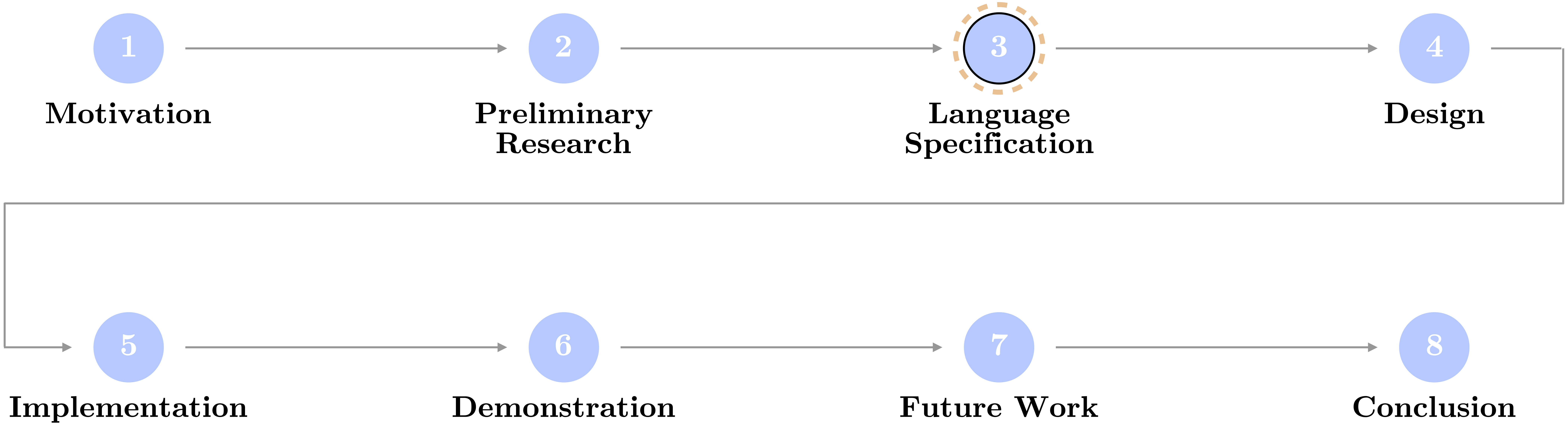
Haskell for Hardware (only software programmers can understand)
Lava (not high-level enough)

Gemini needs to be accessible to both parties

Roadmap

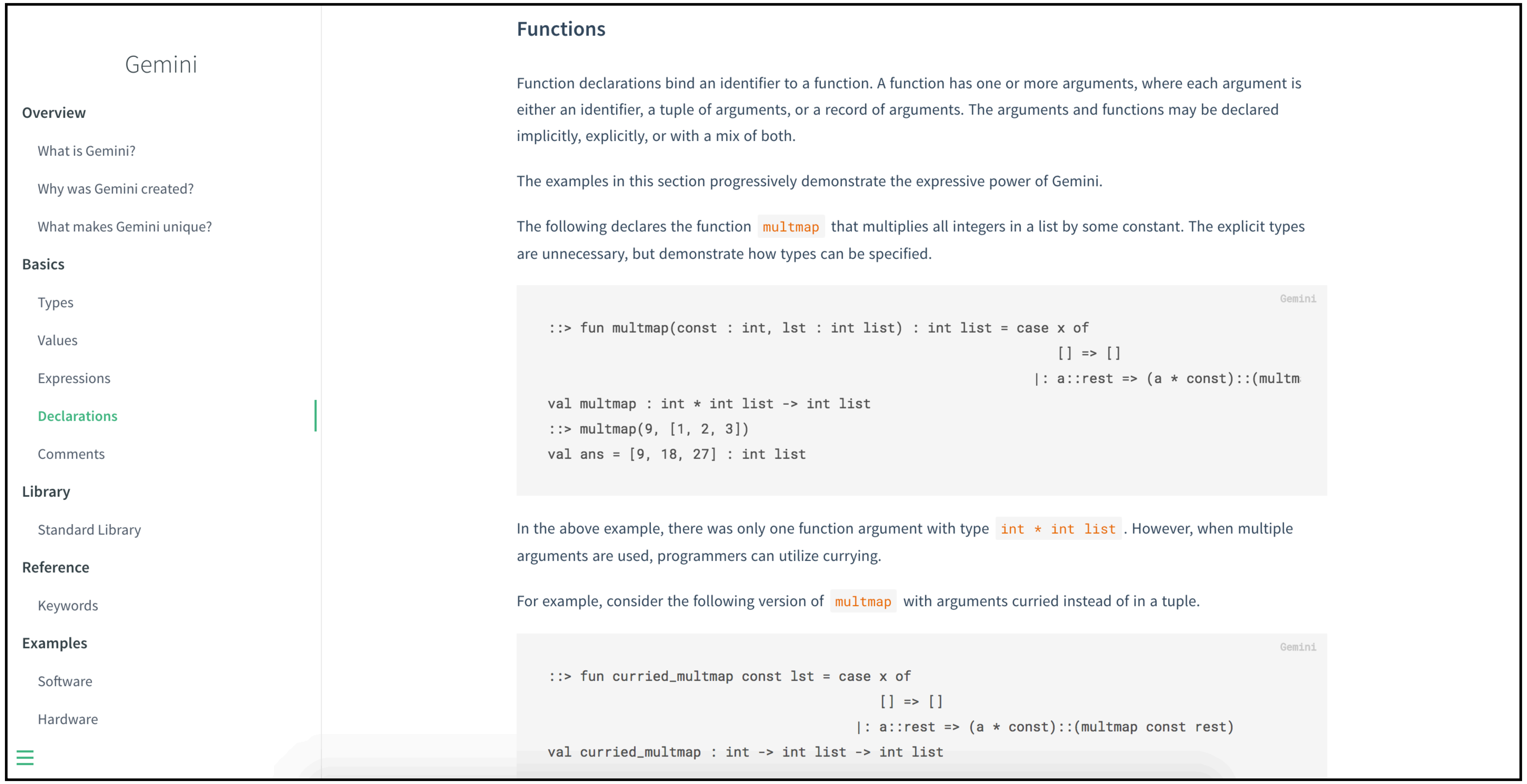

1
Motivation
2
Preliminary Research
3
Language Specification
4
Design
5
Implementation
6
Demonstration
7
Future Work
8
Conclusion

Roadmap


1
Motivation
2
Preliminary Research
3
Language Specification
4
Design
5
Implementation
6
Demonstration
7
Future Work
8
Conclusion

Language Specification

Complete documentation with examples can be found at bit.ly/gemini-docs



Gemini

Functions

Function declarations bind an identifier to a function. A function has one or more arguments, where each argument is either an identifier, a tuple of arguments, or a record of arguments. The arguments and functions may be declared implicitly, explicitly, or with a mix of both.

The examples in this section progressively demonstrate the expressive power of Gemini.

The following declares the function `multmap` that multiplies all integers in a list by some constant. The explicit types are unnecessary, but demonstrate how types can be specified.

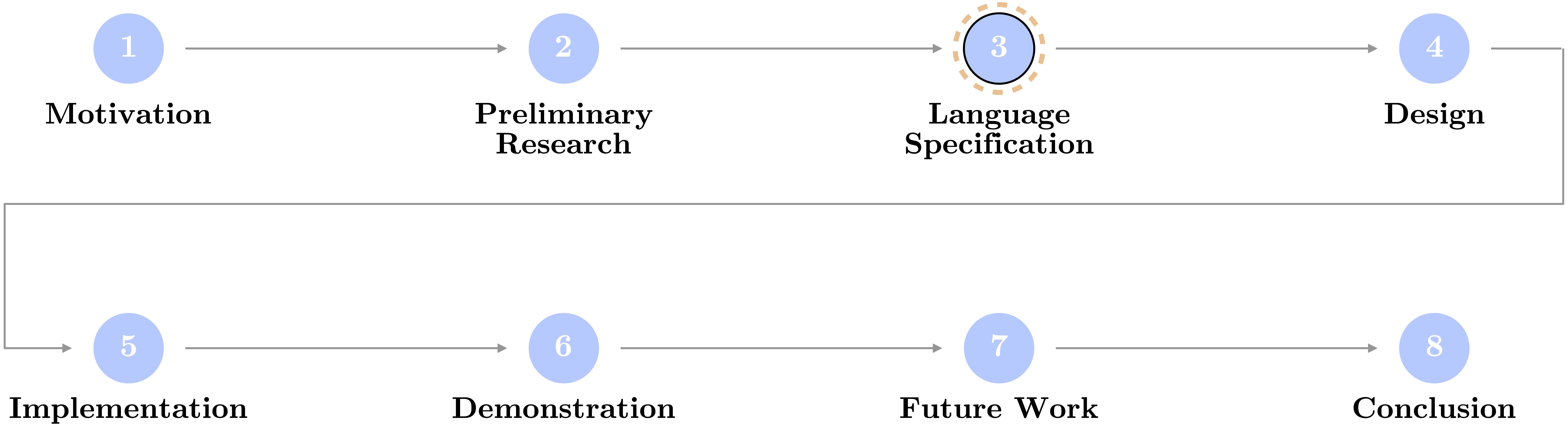
```
Gemini
::> fun multmap(const : int, lst : int list) : int list = case x of
                                                     [] => []
                                                   |: a::rest => (a * const)::(multm
val multmap : int * int list -> int list
::> multmap(9, [1, 2, 3])
val ans = [9, 18, 27] : int list
```

In the above example, there was only one function argument with type `int * int list`. However, when multiple arguments are used, programmers can utilize currying.

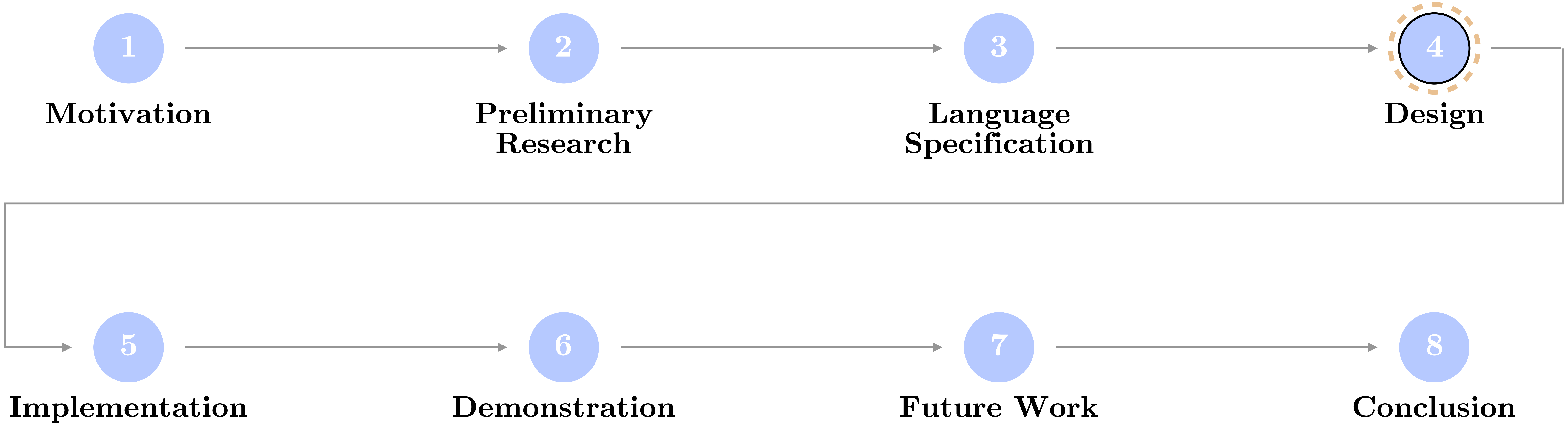
For example, consider the following version of `multmap` with arguments curried instead of in a tuple.

```
Gemini
::> fun curried_multmap const lst = case x of
                                      [] => []
                                    |: a::rest => (a * const)::(multmap const rest)
val curried_multmap : int -> int list -> int list
```

Roadmap


1
Motivation
2
Preliminary Research
3
Language Specification
4
Design
5
Implementation
6
Demonstration
7
Future Work
8
Conclusion

Roadmap


1
Motivation
2
Preliminary Research
3
Language Specification
4
Design
5
Implementation
6
Demonstration
7
Future Work
8
Conclusion

Design

1. Kinding System
2. Grammar
3. Typing Relation
4. Evaluation Rules
5. Proof of Safety

Design

1. Kinding System
2. Grammar
3. Typing Relation
4. Evaluation Rules
5. Proof of Safety

Design // KINDING SYSTEM

What is a type?

DESIGN // KINDING SYSTEM

What is a type?

A classification of a value (`int`, `string`, etc.)

DESIGN // KINDING SYSTEM

What is a type?

A classification of a value (`int`, `string`, etc.)

What is a kind?

DESIGN // KINDING SYSTEM

What is a type?

A classification of a value (`int`, `string`, etc.)

What is a kind?

A classification of a type; “the type of types”

DESIGN // KINDING SYSTEM

In conventional programming languages…

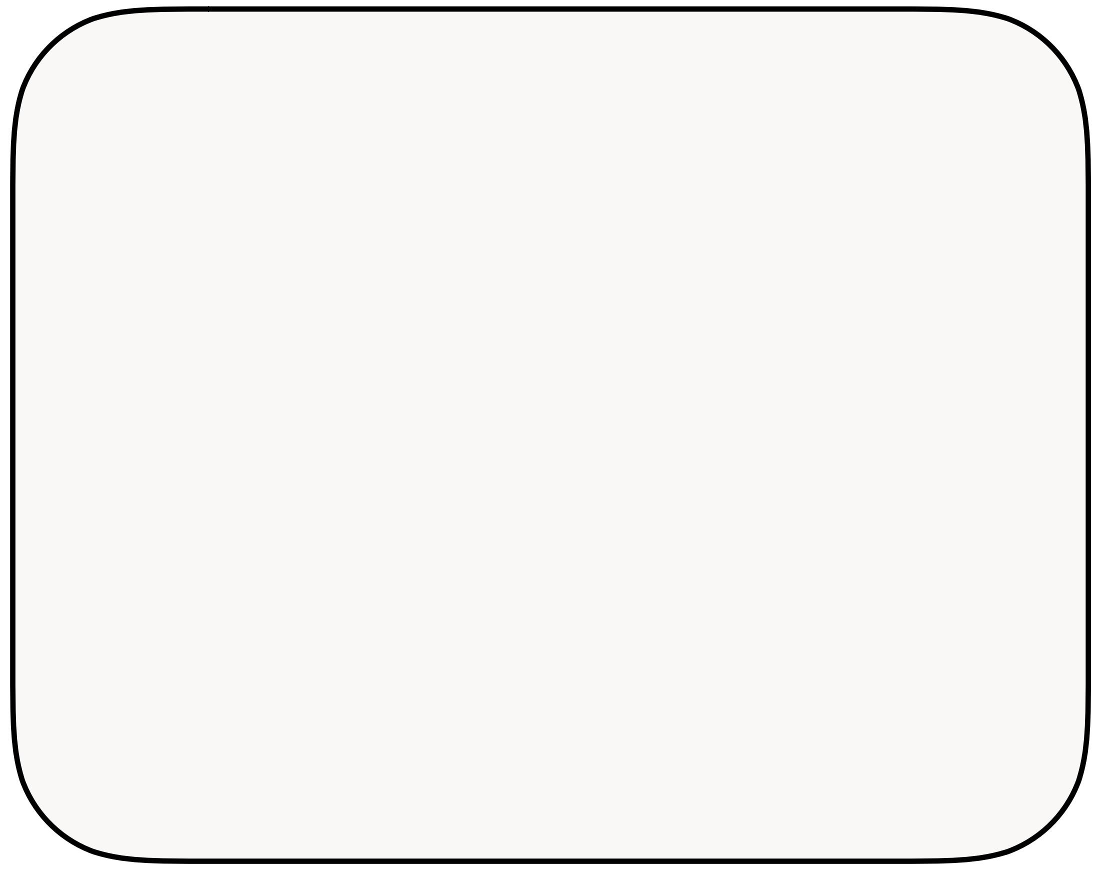
Single atomic kind $*$ (“type”) and the constructor $\Rightarrow$ (“to”)

DESIGN // KINDING SYSTEM

In conventional programming languages…

Single atomic kind $*$ (“type”) and the constructor $\Rightarrow$ (“to”)

$*$



DESIGN // KINDING SYSTEM

In conventional programming languages…

Single atomic kind $*$ (“type”) and the constructor $\Rightarrow$ (“to”)

$*$

$* \Rightarrow *$

Design // KINDING SYSTEM

In conventional programming languages…

Single atomic kind $*$ (“type”) and the constructor $\Rightarrow$ (“to”)

$*$

$* \Rightarrow *$

$* \Rightarrow * \Rightarrow *$

Design // KINDING SYSTEM

In conventional programming languages…

Single atomic kind $*$ (“type”) and the constructor $\Rightarrow$ (“to”)

$*$

`int`

`string`

$* \Rightarrow *$

$* \Rightarrow * \Rightarrow *$

DESIGN // KINDING SYSTEM

In conventional programming languages…

Single atomic kind $*$ (“type”) and the constructor $\Rightarrow$ (“to”)

Design // kinding system

In conventional programming languages…

Single atomic kind $*$ (“type”) and the constructor $\Rightarrow$ (“to”)

`list`

$* \Rightarrow *$

DESIGN // KINDING SYSTEM

In conventional programming languages...

Single atomic kind $*$ ("type") and the constructor $\Rightarrow$ ("to")

`int list`

$* \Rightarrow *$

DESIGN // KINDING SYSTEM

In conventional programming languages…

Single atomic kind $*$ (“type”) and the constructor $\Rightarrow$ (“to”)

$*$

`int`

`string`

$* \Rightarrow *$

`list`

$* \Rightarrow * \Rightarrow *$

`int list`

$*$

DESIGN // KINDING SYSTEM

In conventional programming languages…

Single atomic kind $*$ (“type”) and the constructor $\Rightarrow$ (“to”)

DESIGN // KINDING SYSTEM

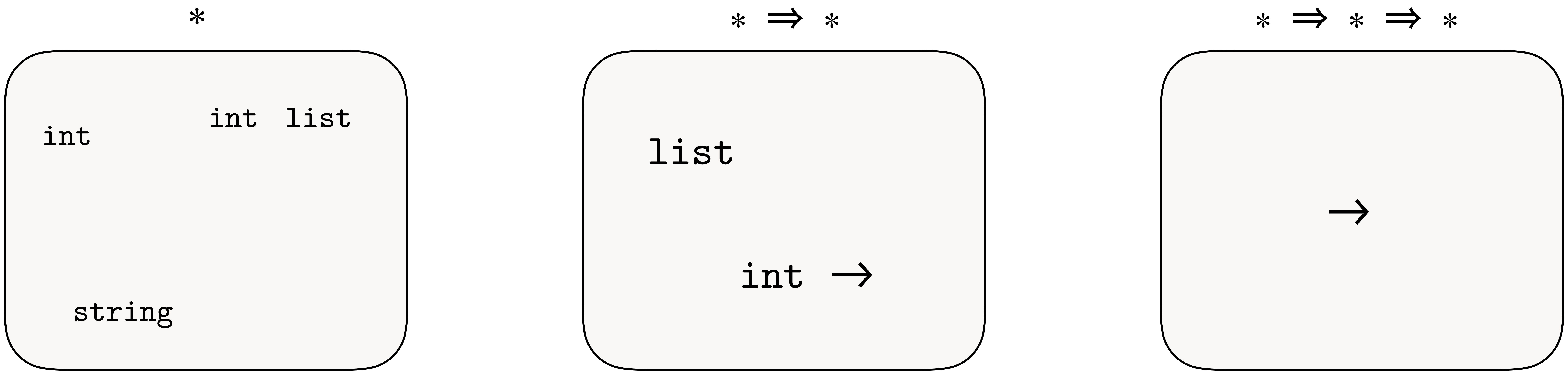
In conventional programming languages...

Single atomic kind $*$ ("type") and the constructor $\Rightarrow$ ("to")

DESIGN // KINDING SYSTEM

In conventional programming languages…

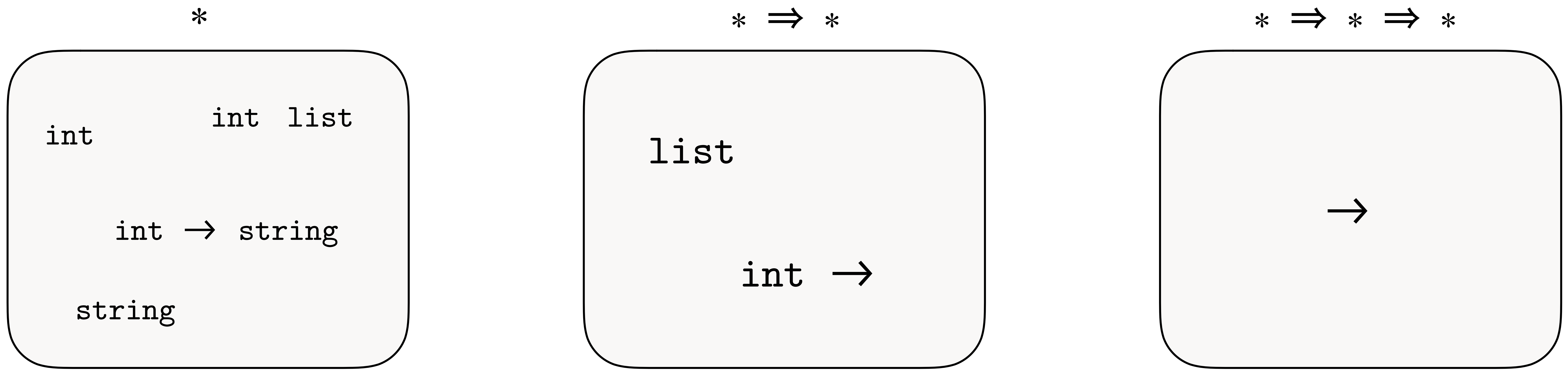
Single atomic kind $*$ (“type”) and the constructor $\Rightarrow$ (“to”)



DESIGN // KINDING SYSTEM

In conventional programming languages…

Single atomic kind $*$ (“type”) and the constructor $\Rightarrow$ (“to”)



DESIGN // KINDING SYSTEM

In Gemini…

Three atomic kinds `S`, `H`, and `M` and the constructor $\Rightarrow$

Design // KINDING SYSTEM

In Gemini...

Three atomic kinds `S`, `H`, and `M` and the constructor $\Rightarrow$

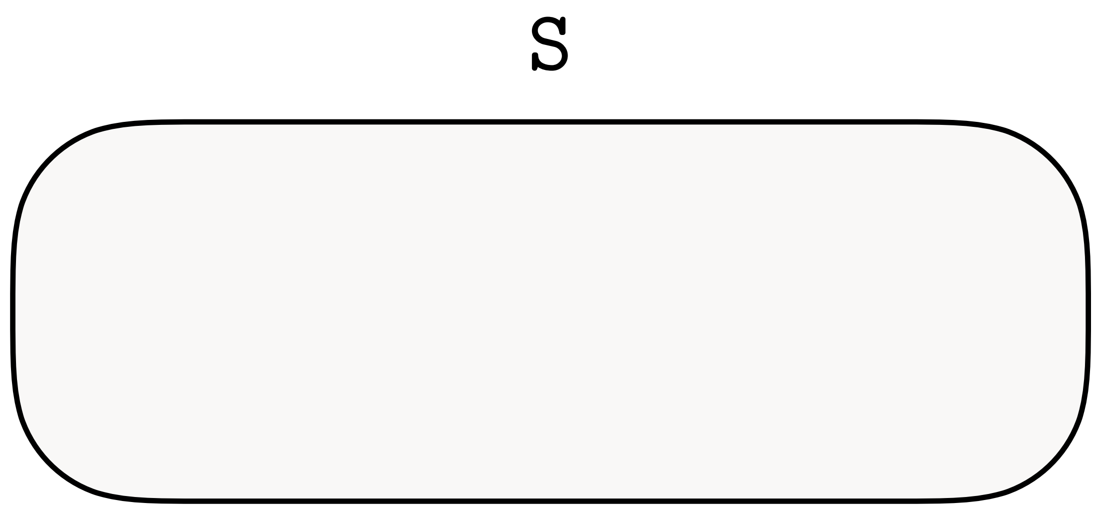
$$
\begin{array}{lll}
S & ::= & \texttt{int} \\
 & | & \texttt{real} \\
 & | & \texttt{string} \\
 & | & T_S\ \texttt{list} \\
 & | & \{\texttt{l}_1\texttt{:}\ T_S\texttt{, ..., l}_n\texttt{:}\ T_S\} \\
 & | & T_S\ \texttt{ref} \\
 & | & T_H\ \texttt{sw} \\
 & | & \texttt{C}_\texttt{i}\ T_S \\
 & | & T_S \rightarrow T_S \\
 & & \\
H & ::= & \texttt{bit} \\
 & | & T_H\ [n] \\
 & | & T_H\ \texttt{@}\ n \\
 & | & \texttt{\#}\{\texttt{l}_1\texttt{:}\ T_H\texttt{, ..., l}_n\texttt{:}\ T_H\} \\
 & & \\
M & ::= & T_H \rightarrow T_H
\end{array}
$$

Design // KINDING SYSTEM

$$
\begin{array}{lll}
S & ::= & \texttt{int} \\
 & | & \texttt{real} \\
 & | & \texttt{string} \\
 & | & T_S\ \texttt{list} \\
 & | & \{\texttt{l}_1\texttt{:}\ T_S\texttt{, ..., l}_n\texttt{:}\ T_S\} \\
 & | & T_S\ \texttt{ref} \\
 & | & T_H\ \texttt{sw} \\
 & | & \texttt{C}_i\ T_S \\
 & | & T_S \rightarrow T_S \\
 & & \\
H & ::= & \texttt{bit} \\
 & | & T_H\ [n] \\
 & | & T_H\ \texttt{@}\ n \\
 & | & \texttt{\#}\{\texttt{l}_1\texttt{:}\ T_H\texttt{, ..., l}_n\texttt{:}\ T_H\} \\
 & & \\
M & ::= & T_H \rightarrow T_H
\end{array}
$$

In Gemini…

Three atomic kinds `S`, `H`, and `M` and the constructor ⇒



DESIGN // KINDING SYSTEM

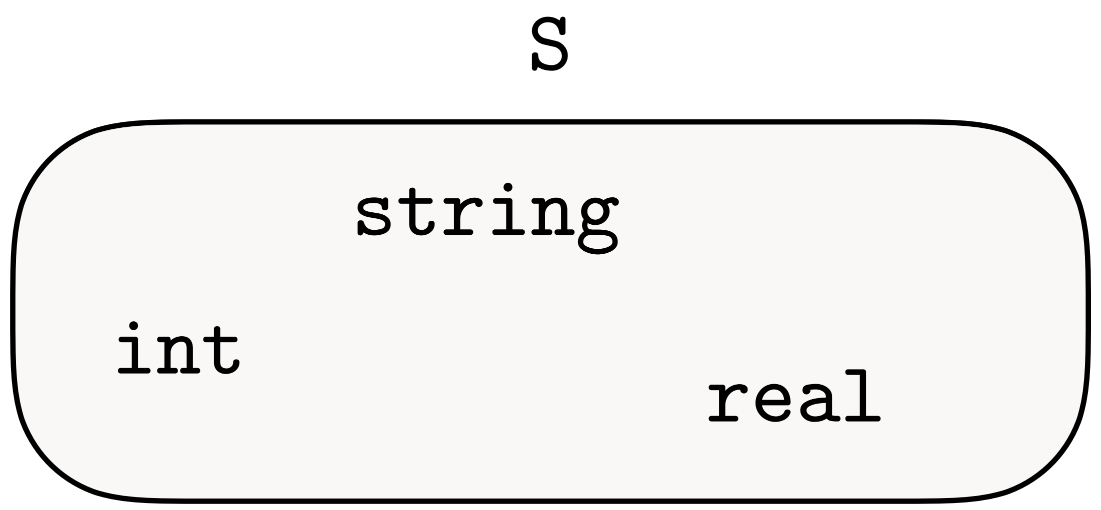
```
S    ::= int
      |  real
      |  string
      |  T_S list
      |  {l_1: T_S, ..., l_n: T_S}
      |  T_S ref
      |  T_H sw
      |  C_i T_S
      |  T_S → T_S

H    ::= bit
      |  T_H [n]
      |  T_H @ n
      |  #{l_1: T_H, ..., l_n: T_H}

M    ::= T_H → T_H
```

In Gemini...

Three atomic kinds `S`, `H`, and `M` and the constructor $\Rightarrow$



DESIGN // KINDING SYSTEM

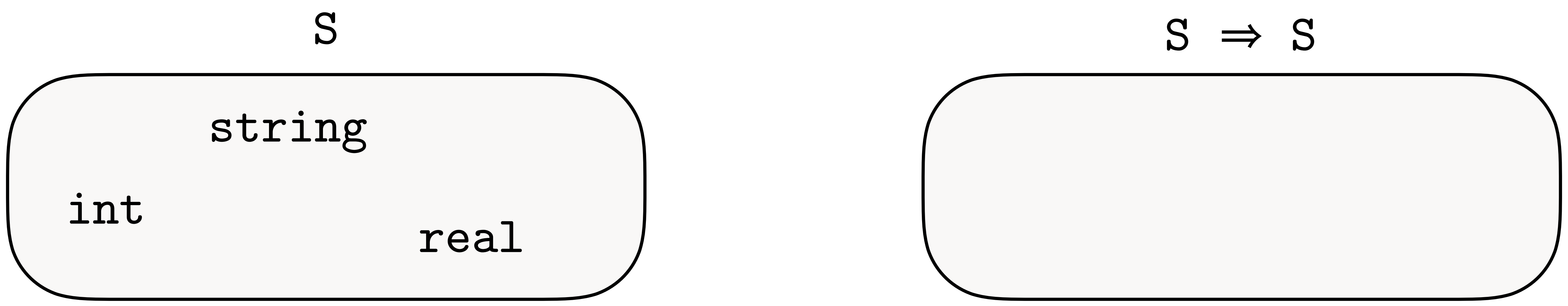
S ::= int
| real
| string
| T_S list
| {l_1: T_S, ..., l_n: T_S}
| T_S ref
| T_H sw
| C_i T_S
| $T_S \to T_S$

H ::= bit
| T_H [n]
| T_H @ n
| #{l_1: T_H, ..., l_n: T_H}

M ::= $T_H \to T_H$

In Gemini...

Three atomic kinds `S`, `H`, and `M` and the constructor $\Rightarrow$



DESIGN // KINDING SYSTEM

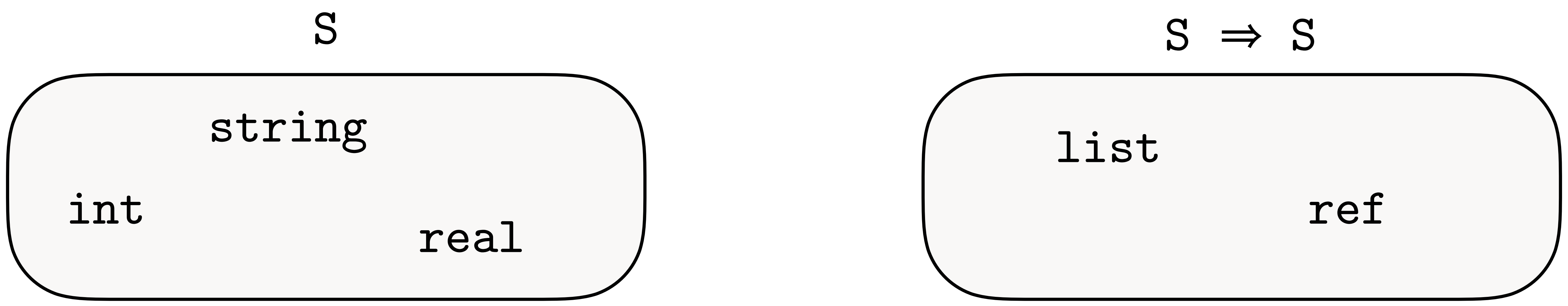
```
S   ::= int
      |  real
      |  string
      |  T_S list
      |  {l₁: T_S, ..., lₙ: T_S}
      |  T_S ref
      |  T_H sw
      |  Cᵢ T_S
      |  T_S → T_S

H   ::= bit
      |  T_H [n]
      |  T_H @ n
      |  #{l₁: T_H, ..., lₙ: T_H}

M   ::= T_H → T_H
```

In Gemini…

Three atomic kinds `S`, `H`, and `M` and the constructor $\Rightarrow$



DESIGN // KINDING SYSTEM

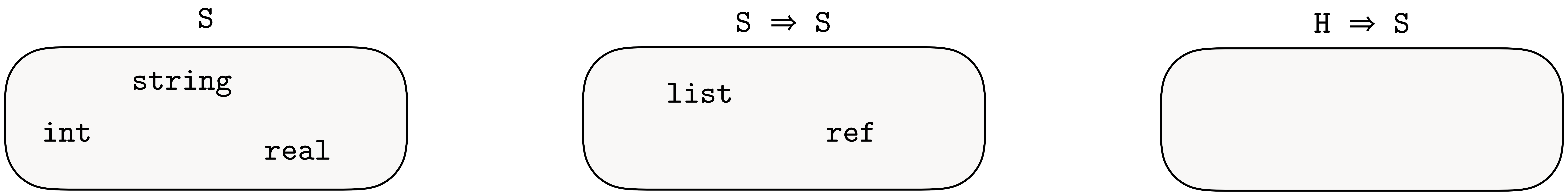
```
S    ::= int
       | real
       | string
       | T_S list
       | {l_1: T_S, ..., l_n: T_S}
       | T_S ref
       | T_H sw
       | C_i T_S
       | T_S → T_S

H    ::= bit
       | T_H [n]
       | T_H @ n
       | #{l_1: T_H, ..., l_n: T_H}

M    ::= T_H → T_H
```

In Gemini…

Three atomic kinds `S`, `H`, and `M` and the constructor ⇒



DESIGN // KINDING SYSTEM

```
S    ::= int
      |  real
      |  string
      |  T_S list
      |  {l₁: T_S, ..., lₙ: T_S}
      |  T_S ref
      |  T_H sw
      |  Cᵢ T_S
      |  T_S → T_S

H    ::= bit
      |  T_H [n]
      |  T_H @ n
      |  #{l₁: T_H, ..., lₙ: T_H}

M    ::= T_H → T_H
```

In Gemini...

Three atomic kinds `S`, `H`, and `M` and the constructor ⇒

`S`: string, int, real

`S ⇒ S`: list, ref

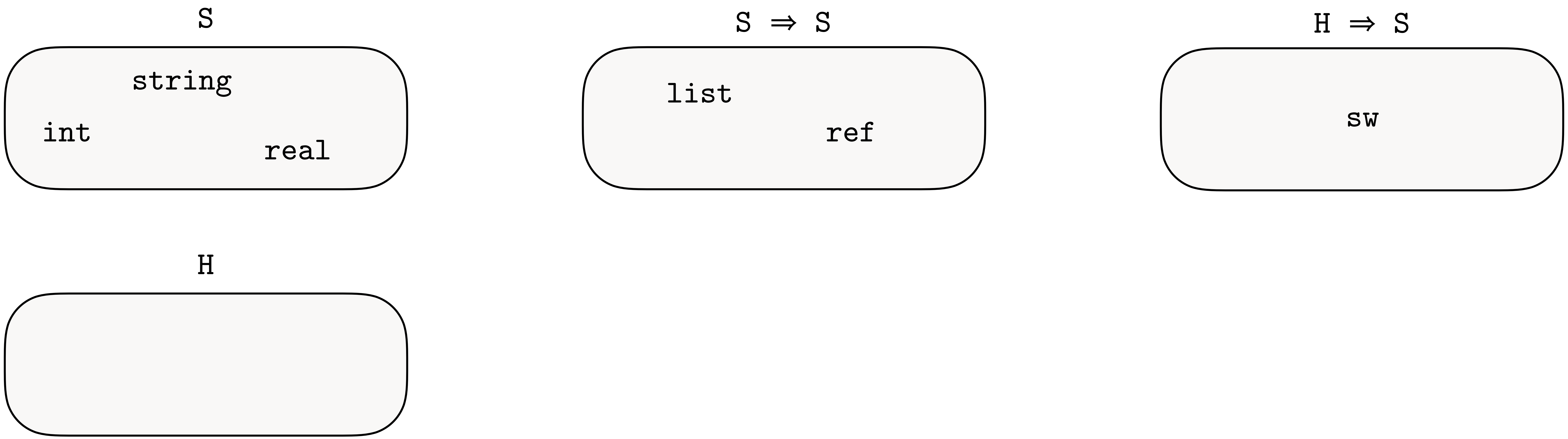
`H ⇒ S`: sw

DESIGN // KINDING SYSTEM

$$
\begin{array}{lll}
S & ::= & \texttt{int} \\
 & | & \texttt{real} \\
 & | & \texttt{string} \\
 & | & T_S\ \texttt{list} \\
 & | & \{\texttt{l}_1\texttt{:}\ T_S\texttt{, ..., l}_n\texttt{:}\ T_S\} \\
 & | & T_S\ \texttt{ref} \\
 & | & T_H\ \texttt{sw} \\
 & | & \texttt{C}_i\ T_S \\
 & | & T_S \rightarrow T_S \\
 & & \\
H & ::= & \texttt{bit} \\
 & | & T_H\ [n] \\
 & | & T_H\ \texttt{@}\ n \\
 & | & \#\{\texttt{l}_1\texttt{:}\ T_H\texttt{, ..., l}_n\texttt{:}\ T_H\} \\
 & & \\
M & ::= & T_H \rightarrow T_H
\end{array}
$$

In Gemini...

Three atomic kinds `S`, `H`, and `M` and the constructor ⇒



DESIGN // KINDING SYSTEM

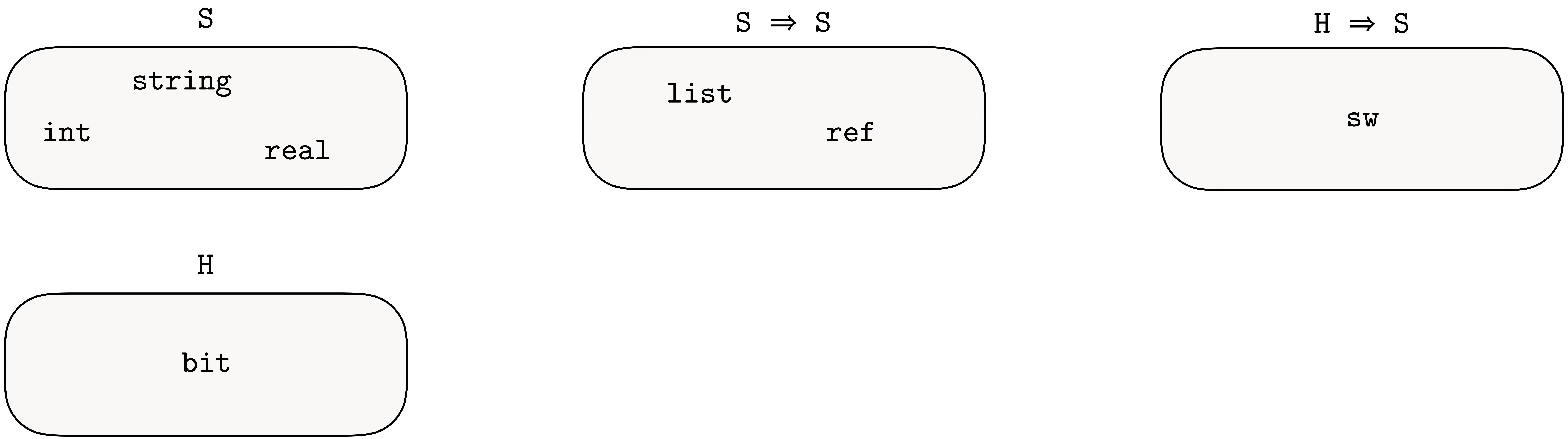
```
S    ::= int
      |  real
      |  string
      |  T_S list
      |  {l_1: T_S, ..., l_n: T_S}
      |  T_S ref
      |  T_H sw
      |  C_i T_S
      |  T_S → T_S

H    ::= bit
      |  T_H [n]
      |  T_H @ n
      |  #{l_1: T_H, ..., l_n: T_H}

M    ::= T_H → T_H
```

In Gemini…

Three atomic kinds `S`, `H`, and `M` and the constructor ⇒



Design // KINDING SYSTEM

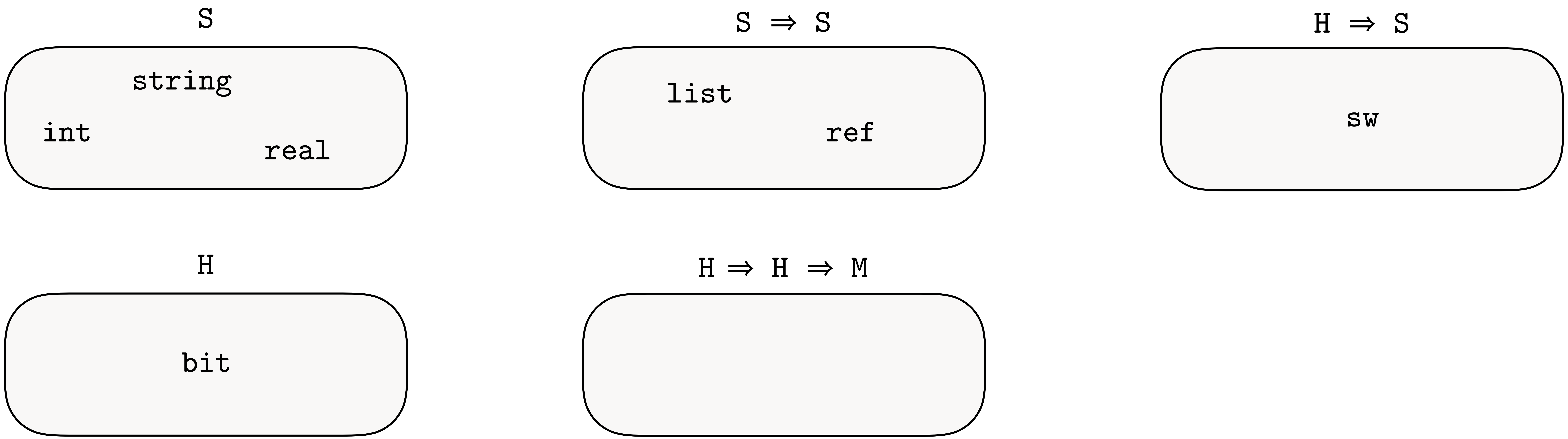
```
S        ::= int
           |  real
           |  string
           |  T_S list
           |  {l₁: T_S, ..., lₙ: T_S}
           |  T_S ref
           |  T_H sw
           |  Cᵢ T_S
           |  T_S → T_S

H        ::= bit
           |  T_H [n]
           |  T_H @ n
           |  #{l₁: T_H, ..., lₙ: T_H}

M        ::= T_H → T_H
```

In Gemini...

Three atomic kinds `S`, `H`, and `M` and the constructor $\Rightarrow$



DESIGN // KINDING SYSTEM

```
S    ::= int
       |  real
       |  string
       |  T_S list
       |  {l₁: T_S, ..., lₙ: T_S}
       |  T_S ref
       |  T_H sw
       |  Cᵢ T_S
       |  T_S → T_S

H    ::= bit
       |  T_H [n]
       |  T_H @ n
       |  #{l₁: T_H, ..., lₙ: T_H}

M    ::= T_H → T_H
```

In Gemini...

Three atomic kinds `S`, `H`, and `M` and the constructor $\Rightarrow$

`S`

`string` `int` `real`

`S ⇒ S`

`list` `ref`

`H ⇒ S`

`sw`

`H`

`bit`

`H ⇒ H ⇒ M`

$\rightsquigarrow$

Design // KINDING SYSTEM

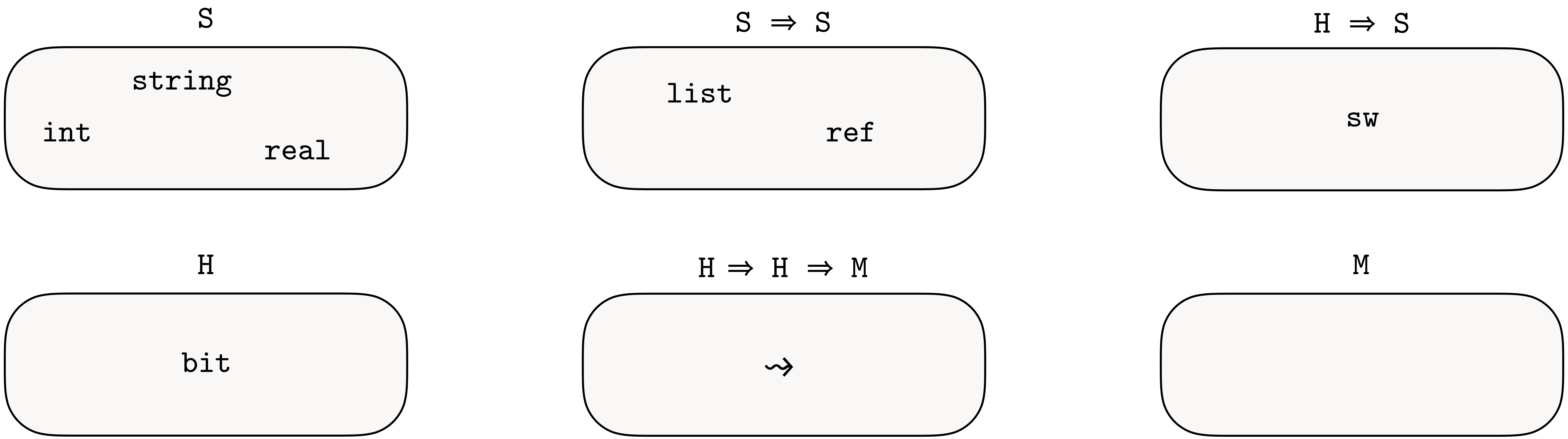
```
S    ::= int
       |  real
       |  string
       |  T_S list
       |  {l₁: T_S, ..., lₙ: T_S}
       |  T_S ref
       |  T_H sw
       |  Cᵢ T_S
       |  T_S → T_S

H    ::= bit
       |  T_H [n]
       |  T_H @ n
       |  #{l₁: T_H, ..., lₙ: T_H}

M    ::= T_H → T_H
```

In Gemini…

Three atomic kinds `S`, `H`, and `M` and the constructor ⇒



Design // KINDING SYSTEM

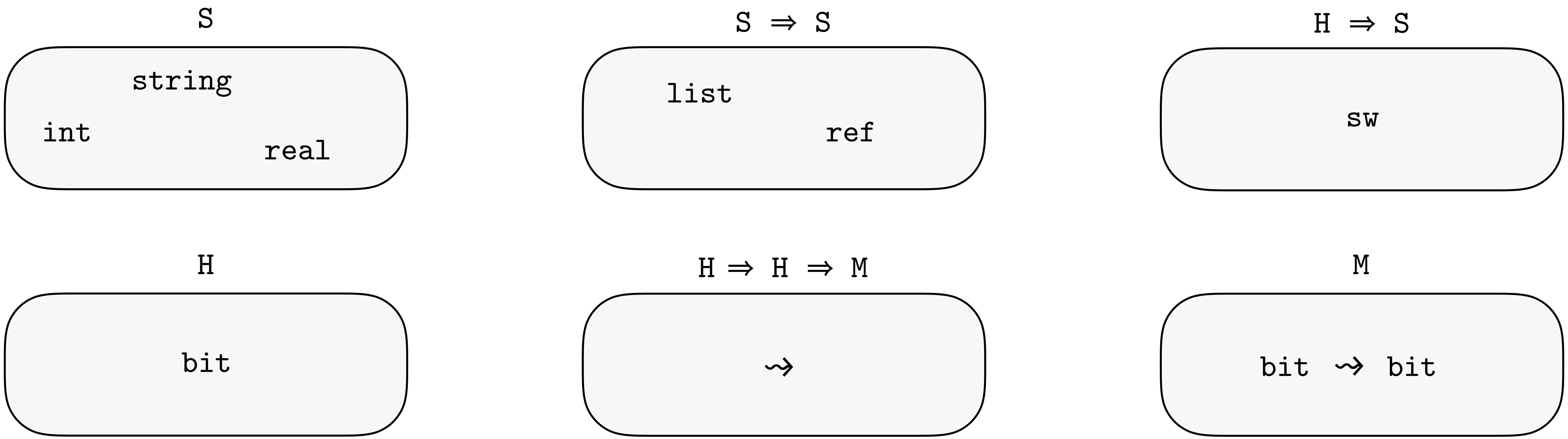
```
S          ::= int
             |  real
             |  string
             |  T_S list
             |  {l₁: T_S, ..., lₙ: T_S}
             |  T_S ref
             |  T_H sw
             |  Cᵢ T_S
             |  T_S → T_S

H          ::= bit
             |  T_H [n]
             |  T_H @ n
             |  #{l₁: T_H, ..., lₙ: T_H}

M          ::= T_H → T_H
```

In Gemini...

Three atomic kinds S, H, and M and the constructor $\Rightarrow$



DESIGN // KINDING SYSTEM

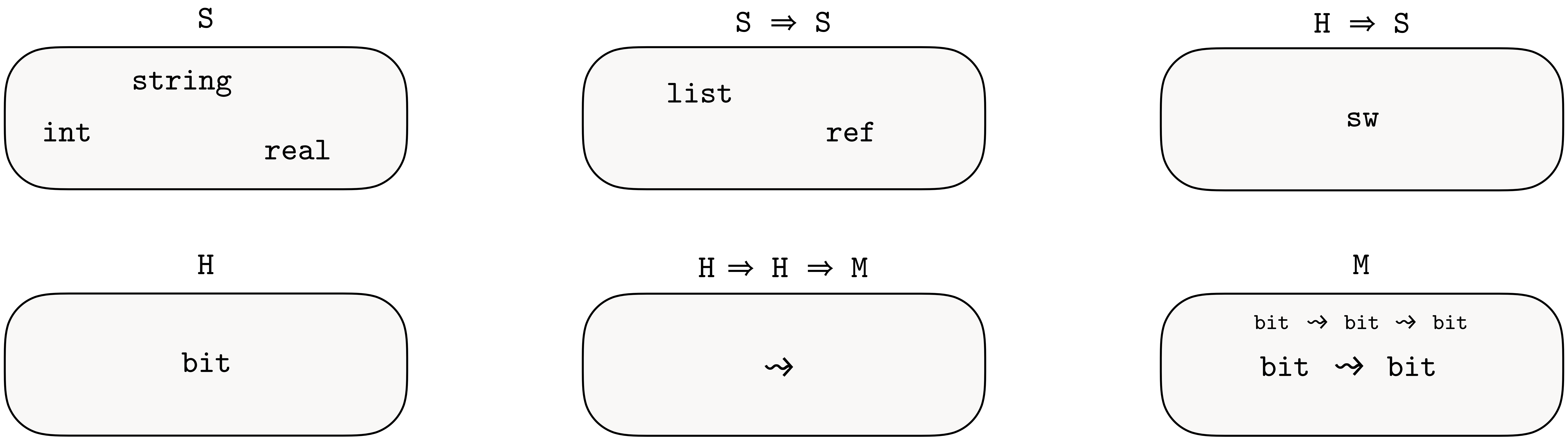
```
S    ::= int
       |  real
       |  string
       |  T_S list
       |  {l₁: T_S, ..., lₙ: T_S}
       |  T_S ref
       |  T_H sw
       |  Cᵢ T_S
       |  T_S → T_S

H    ::= bit
       |  T_H [n]
       |  T_H @ n
       |  #{l₁: T_H, ..., lₙ: T_H}

M    ::= T_H → T_H
```

In Gemini…

Three atomic kinds `S`, `H`, and `M` and the constructor ⇒



DESIGN // KINDING SYSTEM

S ::= int | real | string | T_S list | {l₁: T_S, ..., lₙ: T_S} | T_S ref | T_H sw | Cᵢ T_S | $T_S \to T_S$

H ::= bit | T_H [n] | T_H @ n | #{l₁: T_H, ..., lₙ: T_H}

M ::= $T_H \to T_H$

In Gemini…

Three atomic kinds `S`, `H`, and `M` and the constructor ⇒

S: string, int, real

S ⇒ S: list, ref

H ⇒ S: sw

H: bit

H ⇒ H ⇒ M: ⇝

M: ~~bit ⇝ bit ⇝ bit~~ bit ⇝ bit

DESIGN // KINDING SYSTEM

$$
\begin{array}{lll}
S & ::= & \texttt{int} \\
 & | & \texttt{real} \\
 & | & \texttt{string} \\
 & | & T_S\ \texttt{list} \\
 & | & \{\texttt{l}_1 : T_S, \ldots, \texttt{l}_n : T_S\} \\
 & | & T_S\ \texttt{ref} \\
 & | & T_H\ \texttt{sw} \\
 & | & \texttt{C}_i\ T_S \\
 & | & T_S \rightarrow T_S \\
 & & \\
H & ::= & \texttt{bit} \\
 & | & T_H\ [n] \\
 & | & T_H\ \texttt{@}\ n \\
 & | & \#\{\texttt{l}_1 : T_H, \ldots, \texttt{l}_n : T_H\} \\
 & & \\
M & ::= & T_H \rightarrow T_H
\end{array}
$$

In Gemini...

Three atomic kinds `S`, `H`, and `M` and the constructor $\Rightarrow$

S

`string` ~~`bit list`~~ `int` `real`

S ⇒ S

`list` `ref`

H ⇒ S

`sw`

H

`bit`

H ⇒ H ⇒ M

$\rightsquigarrow$

M

~~`bit` $\rightsquigarrow$ `bit` $\rightsquigarrow$ `bit`~~

`bit` $\rightsquigarrow$ `bit`

Design // KINDING SYSTEM

$$
\begin{array}{lll}
S & ::= & \texttt{int} \\
 & | & \texttt{real} \\
 & | & \texttt{string} \\
 & | & T_S\ \texttt{list} \\
 & | & \{\texttt{l}_1\texttt{:}\ T_S\texttt{,}\ \ldots\texttt{,}\ \texttt{l}_n\texttt{:}\ T_S\} \\
 & | & T_S\ \texttt{ref} \\
 & | & T_H\ \texttt{sw} \\
 & | & \texttt{C}_i\ T_S \\
 & | & T_S \rightarrow T_S \\
H & ::= & \texttt{bit} \\
 & | & T_H\ [n] \\
 & | & T_H\ \texttt{@}\ n \\
 & | & \texttt{\#}\{\texttt{l}_1\texttt{:}\ T_H\texttt{,}\ \ldots\texttt{,}\ \texttt{l}_n\texttt{:}\ T_H\} \\
M & ::= & T_H \rightarrow T_H
\end{array}
$$

In Gemini...

Three atomic kinds `S`, `H`, and `M` and the constructor ⇒

S

`string` ~~`bit list`~~ `int` `real`

S ⇒ S

`list` `ref`

H ⇒ S

`sw`

H

~~`string ref`~~

`bit`

H ⇒ H ⇒ M

⇝

M

~~`bit ⇝ bit ⇝ bit`~~

`bit ⇝ bit`

DESIGN // KINDING SYSTEM

In Gemini...

Three atomic kinds `S`, `H`, and `M` and the constructor $\Rightarrow$

$$
\begin{array}{lcl}
S & ::= & \texttt{int} \\
  & | & \texttt{real} \\
  & | & \texttt{string} \\
  & | & T_S \ \texttt{list} \\
  & | & \{\texttt{l}_1\texttt{:}\ T_S\texttt{, ..., l}_n\texttt{:}\ T_S\} \\
  & | & T_S \ \texttt{ref} \\
  & | & T_H \ \texttt{sw} \\
  & | & \texttt{C}_i \ T_S \\
  & | & T_S \rightarrow T_S \\
  & & \\
H & ::= & \texttt{bit} \\
  & | & T_H \ [n] \\
  & | & T_H \ \texttt{@} \ n \\
  & | & \#\{\texttt{l}_1\texttt{:}\ T_H\texttt{, ..., l}_n\texttt{:}\ T_H\} \\
  & & \\
M & ::= & T_H \rightarrow T_H
\end{array}
$$

DESIGN // KINDING SYSTEM

In Gemini…

Three atomic kinds `S`, `H`, and `M` and the constructor ⇒

Modeling time as a type

$$
\begin{array}{lll}
S & ::= & \texttt{int} \\
 & | & \texttt{real} \\
 & | & \texttt{string} \\
 & | & T_S\ \texttt{list} \\
 & | & \{\texttt{l}_1\texttt{:}\ T_S,\ \ldots,\ \texttt{l}_n\texttt{:}\ T_S\} \\
 & | & T_S\ \texttt{ref} \\
 & | & T_H\ \texttt{sw} \\
 & | & \texttt{C}_i\ T_S \\
 & | & T_S \rightarrow T_S \\
 & & \\
H & ::= & \texttt{bit} \\
 & | & T_H\ [n] \\
 & | & \boxed{T_H\ @\ n} \\
 & | & \#\{\texttt{l}_1\texttt{:}\ T_H,\ \ldots,\ \texttt{l}_n\texttt{:}\ T_H\} \\
 & & \\
M & ::= & T_H \rightarrow T_H
\end{array}
$$

DESIGN // KINDING SYSTEM

In Gemini...

Three atomic kinds `S`, `H`, and `M` and the constructor $\Rightarrow$

Value-parameterized types

$$
\begin{array}{lll}
S & ::= & \texttt{int} \\
 & | & \texttt{real} \\
 & | & \texttt{string} \\
 & | & T_S\ \texttt{list} \\
 & | & \{\texttt{l}_1\texttt{:}\ T_S\texttt{,}\ \ldots\texttt{,}\ \texttt{l}_n\texttt{:}\ T_S\} \\
 & | & T_S\ \texttt{ref} \\
 & | & T_H\ \texttt{sw} \\
 & | & \texttt{C}_\texttt{i}\ T_S \\
 & | & T_S \rightarrow T_S \\
 & & \\
H & ::= & \texttt{bit} \\
 & | & \boxed{T_H\ [n]} \\
 & | & T_H\ \texttt{@}\ n \\
 & | & \texttt{\#}\{\texttt{l}_1\texttt{:}\ T_H\texttt{,}\ \ldots\texttt{,}\ \texttt{l}_n\texttt{:}\ T_H\} \\
 & & \\
M & ::= & T_H \rightarrow T_H
\end{array}
$$

Design

1. Kinding System
2. Grammar
3. Typing Relation
4. Evaluation Rules
5. Proof of Safety

DESIGN // GRAMMAR

⟨*exp*⟩ ::= ⟨*literal*⟩
| ⟨*access*⟩
| ⟨*let binding*⟩
| ⟨*conditional*⟩
| ⟨*operation*⟩
| ⟨*assignment*⟩
| ⟨*pattern match*⟩
| ⟨*sequence*⟩
| ⟨*application*⟩

⟨*literal*⟩ ::= ⟨*identifier*⟩
| ⟨*integer literal*⟩
| ⟨*real literal*⟩
| ⟨*string literal*⟩
| ⟨*list literal*⟩
| ⟨*software record literal*⟩
| ⟨*ref literal*⟩
| ⟨*sw literal*⟩

⟨*identifier*⟩ ::= ⟨*id-start*⟩ ⟨*id-tail*⟩

⟨*id-start*⟩ ::= {any alphabetic character or underscore}

⟨*id-tail*⟩ ::= {any alphanumeric character or underscore} ⟨*id-tail*⟩
| ε

⟨*integer literal*⟩ ::= ⟨*binary-integer*⟩
| ⟨*octal-integer*⟩
| ⟨*decimal-integer*⟩
| ⟨*hex-integer*⟩

⟨*binary-integer*⟩ ::= `#'b:` ⟨*binary-digits*⟩

⟨*octal-integer*⟩ ::= `#'o:` ⟨*octal-digits*⟩

⟨*decimal-integer*⟩ ::= ⟨*decimal-digits*⟩
| ⟨*sign*⟩ ⟨*decimal-digits*⟩

⟨*hex-integer*⟩ ::= `#'x:` ⟨*hex-digits*⟩

Design // GRAMMAR

⟨*exp*⟩	::=	⟨*literal*⟩
	|	⟨*access*⟩
	|	⟨*let binding*⟩
	|	⟨*conditional*⟩
	|	⟨*operation*⟩
	|	⟨*assignment*⟩
	|	⟨*pattern match*⟩
	|	⟨*sequence*⟩
	|	⟨*application*⟩
⟨*literal*⟩	::=	⟨*identifier*⟩
	|	⟨*integer literal*⟩
	|	⟨*real literal*⟩
	|	⟨*string literal*⟩
	|	⟨*list literal*⟩
	|	⟨*software record literal*⟩
	|	⟨*ref literal*⟩
	|	⟨*sw literal*⟩

⟨*identifier*⟩	::=	⟨*id-start*⟩ ⟨*id-tail*⟩
⟨*id-start*⟩	::=	{any alphabetic character or underscore}
⟨*id-tail*⟩	::=	{any alphanumeric character or underscore} ⟨*id-tail*⟩
	|	ε
⟨*integer literal*⟩	::=	⟨*binary-integer*⟩
	|	⟨*octal-integer*⟩
	|	⟨*decimal-integer*⟩
	|	⟨*hex-integer*⟩
⟨*binary-integer*⟩	::=	`#'b:` ⟨*binary-digits*⟩
⟨*octal-integer*⟩	::=	`#'o:` ⟨*octal-digits*⟩
⟨*decimal-integer*⟩	::=	⟨*decimal-digits*⟩
	|	⟨*sign*⟩ ⟨*decimal-digits*⟩
⟨*hex-integer*⟩	::=	`#'x:` ⟨*hex-digits*⟩

Shown in full in Appendix B

DESIGN // GRAMMAR

```
⟨exp⟩                     ::= ⟨literal⟩
                            | ⟨access⟩
                            | ⟨let binding⟩
                            | ⟨operation⟩
                            | ⟨parameterization⟩

⟨literal⟩                 ::= ⟨bit literal⟩
                            | ⟨array literal⟩
                            | ⟨hardware record literal⟩

⟨bit literal⟩             ::= 'b: ⟨binary-digit⟩

⟨array literal⟩           ::= #[ ⟨list-body⟩ ]
                            | ⟨gen-array⟩
                            | ⟨bit-array⟩

⟨gen-array⟩               ::= #[ ⟨exp⟩ ; gen ⟨identifier⟩ => ⟨exp⟩ ]

⟨bit-array⟩               ::= ⟨exp⟩ 's: ⟨exp⟩
                            | ⟨exp⟩ 'u: ⟨exp⟩
                            | ⟨exp⟩ 'r: ⟨exp⟩

⟨hardware record literal⟩ ::= #{ ⟨record-body⟩ }
```

Design // GRAMMAR

⟨*exp*⟩	::=	⟨*literal*⟩
	|	⟨*access*⟩
	|	⟨*let binding*⟩
	|	⟨*operation*⟩
	|	⟨*parameterization*⟩
⟨*literal*⟩	::=	⟨*bit literal*⟩
	|	⟨*array literal*⟩
	|	⟨*hardware record literal*⟩
⟨*bit literal*⟩	::=	`'b:` ⟨*binary-digit*⟩
⟨*array literal*⟩	::=	`#[` ⟨*list-body*⟩ `]`
	|	⟨*gen-array*⟩
	|	⟨*bit-array*⟩
⟨*gen-array*⟩	::=	`#[` ⟨*exp*⟩ `; gen` ⟨*identifier*⟩ `=>` ⟨*exp*⟩ `]`
⟨*bit-array*⟩	::=	⟨*exp*⟩ `'s:` ⟨*exp*⟩
	|	⟨*exp*⟩ `'u:` ⟨*exp*⟩
	|	⟨*exp*⟩ `'r:` ⟨*exp*⟩
⟨*hardware record literal*⟩	::=	`#{` ⟨*record-body*⟩ `}`

Shown in full in Appendix B

Design

1. Kinding System
2. Grammar
3. Typing Relation
4. Evaluation Rules
5. Proof of Safety

DESIGN // TYPING RELATION

Each typing rule is a theorem

DESIGN // TYPING RELATION

Each typing rule is a theorem

$$\frac{\text{hypothesis}}{\text{conclusion}}\ (\text{T-NAME})$$

Design // typing relation

Examples of typing rules

$$\frac{\texttt{t}_1\texttt{:int} \quad \texttt{t}_2\texttt{:int}}{\texttt{t}_1 \texttt{ + } \texttt{t}_2 \texttt{ : int}} \ (\textsc{T-Int-Add})$$

DESIGN // TYPING RELATION

Examples of typing rules

$$\frac{\texttt{t}_1 : \texttt{T}_\texttt{H} \qquad \texttt{t}_2 : \texttt{T}_\texttt{H}}{\texttt{t}_1 \ \& \ \texttt{t}_2 : \texttt{T}_\texttt{H}} \quad \text{(T-AND)}$$

DESIGN // TYPING RELATION

Examples of typing rules

$$\frac{\texttt{t}_1 : \texttt{T}_\texttt{H} \qquad \texttt{t}_2 : \texttt{T}_\texttt{H}}{\texttt{t}_1 \ \& \ \texttt{t}_2 : \texttt{T}_\texttt{H}} \quad \text{(T-AND)}$$

51 rules in total, shown in full in Appendix B

Design

1. Kinding System
2. Grammar
3. Typing Relation
4. Evaluation Rules
5. Proof of Safety

DESIGN // EVALUATION RULES

Defining the semantics of the language

DESIGN // EVALUATION RULES

Defining the semantics of the language

1. Operational semantics
2. Denotational semantics
3. Axiomatic semantics

Design // evaluation rules

Defining the semantics of the language

1. Operational semantics
2. ~~Denotational semantics~~
3. ~~Axiomatic semantics~~

DESIGN // EVALUATION RULES

Operational semantics define an abstract state machine

Design // EVALUATION RULES

Operational semantics define an abstract state machine

$$f\left(\boxed{\texttt{exp}}\right) = \boxed{\texttt{exp'}}$$

DESIGN // EVALUATION RULES

Operational semantics define an abstract state machine

$$f\left(\boxed{\texttt{exp}}\right) = \boxed{\texttt{exp'}}$$

$$f\left(\boxed{\texttt{val}}\right) = \texttt{terminal}$$

DESIGN // EVALUATION RULES

Operational semantics can be further partitioned

1. Structural (small-step)
2. Natural (big-step)

DESIGN // EVALUATION RULES

Operational semantics can be further partitioned

1. Structural (small-step)
2. ~~Natural (big-step)~~

Design // evaluation rules

Each evaluation rule is a theorem

$$\frac{\text{hypothesis}}{\text{conclusion}}\ \text{(E-NAME)}$$

DESIGN // EVALUATION RULES

Examples of evaluation rules

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{if}\ \mathtt{t_1}\ \mathtt{then}\ \mathtt{t_2}\ \mathtt{else}\ \mathtt{t_3} \longrightarrow \mathtt{if}\ \mathtt{t_1'}\ \mathtt{then}\ \mathtt{t_2}\ \mathtt{else}\ \mathtt{t_3}} \quad \text{(E-IFELSE)}$$

Design // EVALUATION RULES

Examples of evaluation rules

$$\texttt{if 0 then } \mathtt{t}_2 \texttt{ else } \mathtt{t}_3 \longrightarrow \mathtt{t}_3 \qquad \text{(E-IfElse-F)}$$

DESIGN // EVALUATION RULES

Examples of evaluation rules

$$\texttt{if 0 then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{t}_3 \qquad \text{(E-IFELSE-F)}$$

97 rules in total, shown in full in Appendix B

DESIGN

1. Kinding System
2. Grammar
3. Typing Relation
4. Evaluation Rules
5. Proof of Safety

DESIGN // PROOF OF SAFETY

We first prove two supporting theorems

DESIGN // PROOF OF SAFETY

We first prove two supporting theorems

Theorem of Progress

Suppose `t` is a closed, well-typed term ($\vdash$ `t : T` for some `T`). Then either `t` is a value or else there is some `t'` with `t` $\longrightarrow$ `t'`.

DESIGN // PROOF OF SAFETY

We first prove two supporting theorems

Theorem of Progress

Suppose `t` is a closed, well-typed term ($\vdash$ `t : T` for some `T`). Then either `t` is a value or else there is some `t'` with `t` $\longrightarrow$ `t'`.

Theorem of Preservation

If `t : R` and `t` $\longrightarrow$ `t'`, then `t' : R`.

DESIGN // PROOF OF SAFETY

Proof by structural induction

Proof: By structural induction on a derivation of $\mathtt{t} : \mathtt{T}$.

Case T-INT, T-REAL, T-STRING, T-BIT, T-NIL:
Immediate since t is a value.

Case T-APP:

$\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$
$\vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12}$
$\vdash \mathtt{t}_2 : \mathtt{T}_{12}$

By the induction hypothesis, either $\mathtt{t}_1$ is a value or else there is some other $\mathtt{t}_1'$ for which $\mathtt{t}_1 \longrightarrow \mathtt{t}_1'$, and likewise for $\mathtt{t}_2$. If $\mathtt{t}_1 \longrightarrow \mathtt{t}_1'$ then by E-APP1, $\mathtt{t} \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2$. On the other hand, if $\mathtt{t}_1$ is a value and $\mathtt{t}_2 \longrightarrow \mathtt{t}_2'$, then by E-APP2, $\mathtt{t} \longrightarrow \mathtt{t}_1\ \mathtt{t}_2'$. Finally, if both $\mathtt{t}_1$ and $\mathtt{t}_2$ are values, then case 5 of the canonical forms lemma tells us that $\mathtt{t}_1$ has the form $\lambda x : \mathtt{T}_{11}.\mathtt{t}_{12}$ and so by E-APPABS, $\mathtt{t} \longrightarrow [\mathtt{x} \mapsto \mathtt{t}_2]\mathtt{t}_{12}$ which is a value.

Proof: By structural induction on a derivation of $\mathtt{t} : \mathtt{T}$. At each step of the induction, we assume that the desired property holds for all subderivations (i.e. that if $\mathtt{s} : \mathtt{S}$ and $\mathtt{s} \longrightarrow \mathtt{s}'$, then $\mathtt{s}' : \mathtt{S}$, whenever $\mathtt{s} : \mathtt{S}$ is proved by a subderivation of the present one) and proceed by case analysis on the final rule in the derivation.

Case T-VAR:

$\mathtt{t} = \mathtt{x}$
$\mathtt{x} : \mathtt{T} \in \Gamma$

If the last rule in the derivation is T-VAR, then we know from the form of this rule that t must be a variable of type T. Thus t is a value, so it cannot be the case that $\mathtt{t} \longrightarrow \mathtt{t}'$ for any $\mathtt{t}'$, and the requirements of the theorem are vacuously satisfied.

Case T-APP:

$\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$
$\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12}$
$\Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}$
$\mathtt{T} = \mathtt{T}_{12}$

Looking at the evaluation rules with application on the left-hand side, we find that there are three rules by which $\mathtt{t} \longrightarrow \mathtt{t}'$ can be derived: E-APP1, E-APP2, and E-APPABS. We consider each case separately.

Subcase E-APP1:

$\mathtt{t}_1 \longrightarrow \mathtt{t}_1'$
$\mathtt{t}' = \mathtt{t}_1'\ \mathtt{t}_2$

From the assumptions of the T-APP case, we have a subderivation of the original typing derivation whose conclusion is $\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12}$. We can apply the induction hypothesis to this subderivation obtaining $\Gamma \vdash \mathtt{t}_1' : \mathtt{T}_{11} \to \mathtt{T}_{12}$. Combining this with the fact that $\Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}$, we can apply rule T-APP to conclude that $\Gamma \vdash \mathtt{t}' : \mathtt{T}$.

Subcase E-APP2:
Similar to E-APP1.

Subcase E-APPABS:

$\mathtt{t}_1 = \lambda x : \mathtt{T}_{11}.\mathtt{t}_{12}$
$\mathtt{t}_2 = \mathtt{v}_2$
$\mathtt{t}' = [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12}$

DESIGN // PROOF OF SAFETY

Proof by structural induction

Proof: By structural induction on a derivation of t : T.

Case T-INT, T-REAL, T-STRING, T-BIT, T-NIL:
Immediate since t is a value.

Case T-APP:
$t = t_1\ t_2$
$\vdash t_1 : T_{11} \to T_{12}$
$\vdash t_2 : T_{12}$

By the induction hypothesis, either t_1 is a value or else there is some other t_1' for which $t_1 \longrightarrow t_1'$, and likewise for t_2. If $t_1 \longrightarrow t_1'$ then by E-APP1, $t \longrightarrow t_1'\ t_2$. On the other hand, if t_1 is a value and $t_2 \longrightarrow t_2'$, then by E-APP2, $t \longrightarrow t_1\ t_2'$. Finally, if both t_1 and t_2 are values, then case 5 of the canonical forms lemma tells us that t_1 has the form $\lambda x : T_{11}.t_{12}$ and so by E-APPABS, $t \longrightarrow [x \mapsto t_2]t_{12}$ which is a value.

Shown in full in Appendix B

Proof: By structural induction on a derivation of t : T. At each step of the induction, we assume that the desired property holds for all subderivations (i.e. that if s : S and $s \longrightarrow s'$, then s' : S, whenever s : S is proved by a subderivation of the present one) and proceed by case analysis on the final rule in the derivation.

Case T-VAR:
$t = x$
$x : T \in \Gamma$

If the last rule in the derivation is T-VAR, then we know from the form of this rule that t must be a variable of type T. Thus t is a value, so it cannot be the case that $t \longrightarrow t'$ for any t', and the requirements of the theorem are vacuously satisfied.

Case T-APP:
$t = t_1\ t_2$
$\Gamma \vdash t_1 : T_{11} \to T_{12}$
$\Gamma \vdash t_2 : T_{11}$
$T = T_{12}$

Looking at the evaluation rules with application on the left-hand side, we find that there are three rules by which $t \longrightarrow t'$ can be derived: E-APP1, E-APP2, and E-APPABS. We consider each case separately.

Subcase E-APP1:
$t_1 \longrightarrow t_1'$
$t' = t_1'\ t_2$

From the assumptions of the T-APP case, we have a subderivation of the original typing derivation whose conclusion is $\Gamma \vdash t_1 : T_{11} \to T_{12}$. We can apply the induction hypothesis to this subderivation obtaining $\Gamma \vdash t_1' : T_{11} \to T_{12}$. Combining this with the fact that $\Gamma \vdash t_2 : T_{11}$, we can apply rule T-APP to conclude that $\Gamma \vdash t' : T$.

Subcase E-APP2:
Similar to E-APP1.

Subcase E-APPABS:
$t_1 = \lambda x : T_{11}.t_{12}$
$t_2 = v_2$
$t' = [x \mapsto v_2]t_{12}$

// PROOF OF SAFETY

Definition

A term is in a stuck state if no evaluation rules apply to it but it is not a value.

DESIGN // PROOF OF SAFETY

Definition

A term is in a stuck state if no evaluation rules apply to it but it is not a value.

Theorem of Safety

A well-typed term can never reach a stuck state during evaluation.

Design // PROOF OF SAFETY

Definition

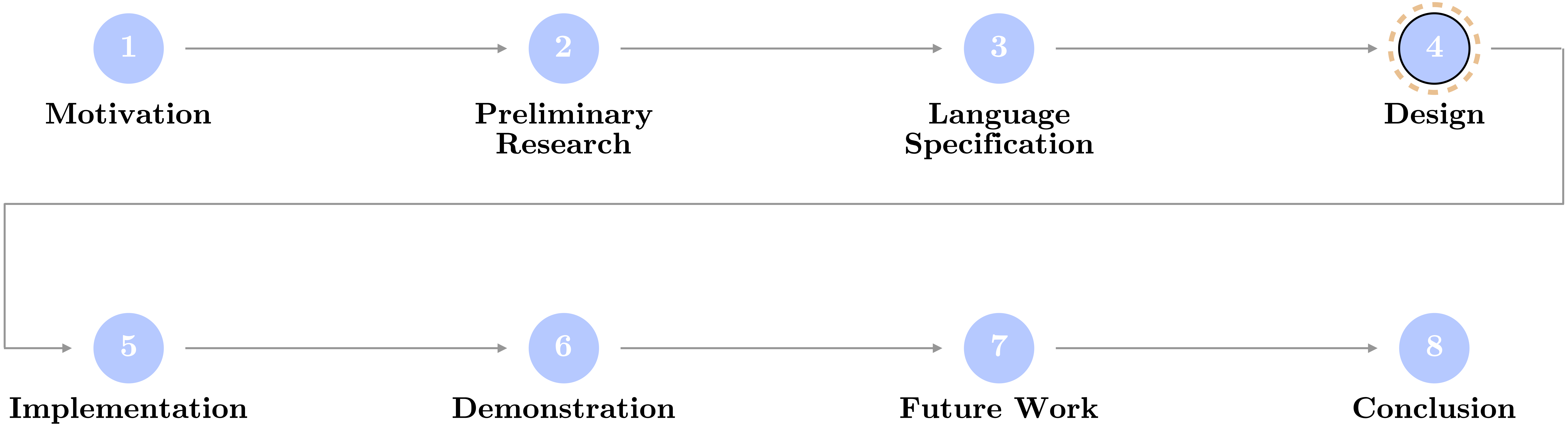
A term is in a stuck state if no evaluation rules apply to it but it is not a value.

Theorem of Safety

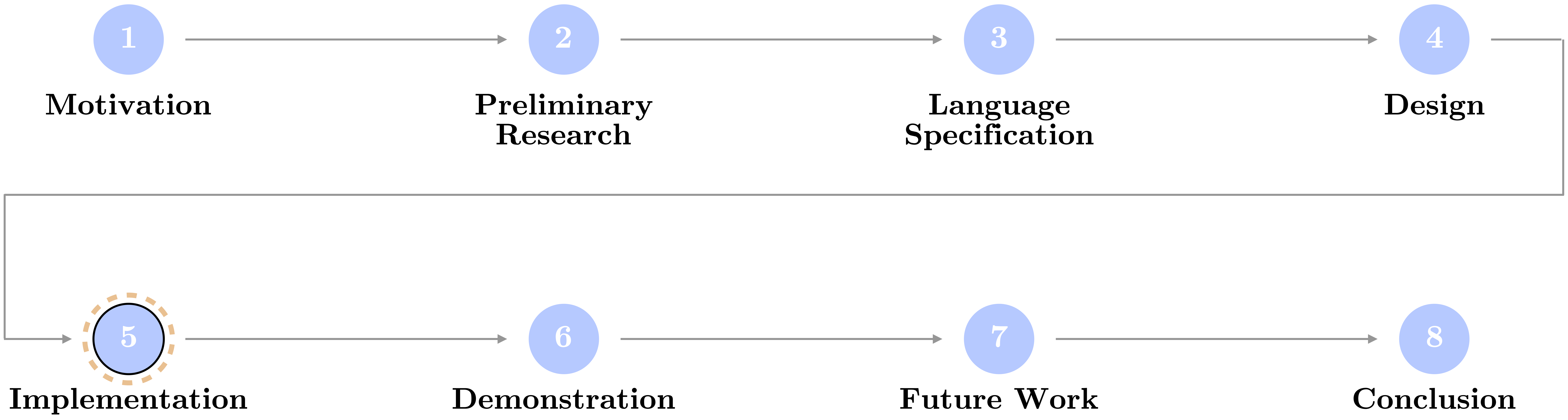
A well-typed term can never reach a stuck state during evaluation.

Proof: Progress tells us a well-typed term can either always take a step of evaluation or it is already a value. Preservation tells us if a well-typed term takes a step of evaluation, the resulting term is also well-typed. In combination and inductively, these guarantee safety. $\square$

Roadmap

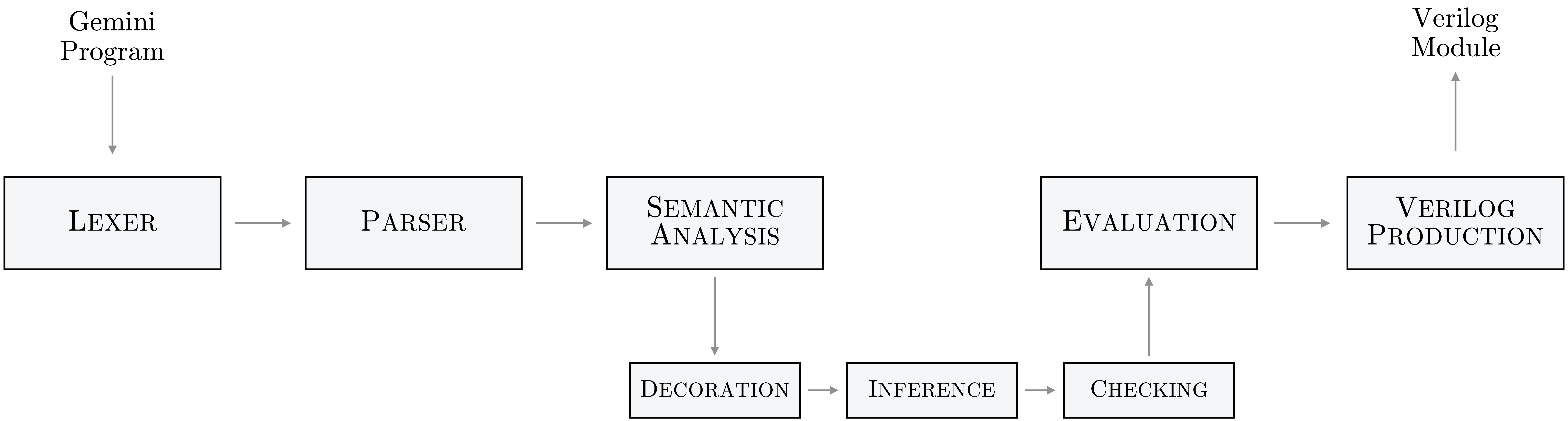

1
Motivation
2
Preliminary Research
3
Language Specification
4
Design
5
Implementation
6
Demonstration
7
Future Work
8
Conclusion

Roadmap

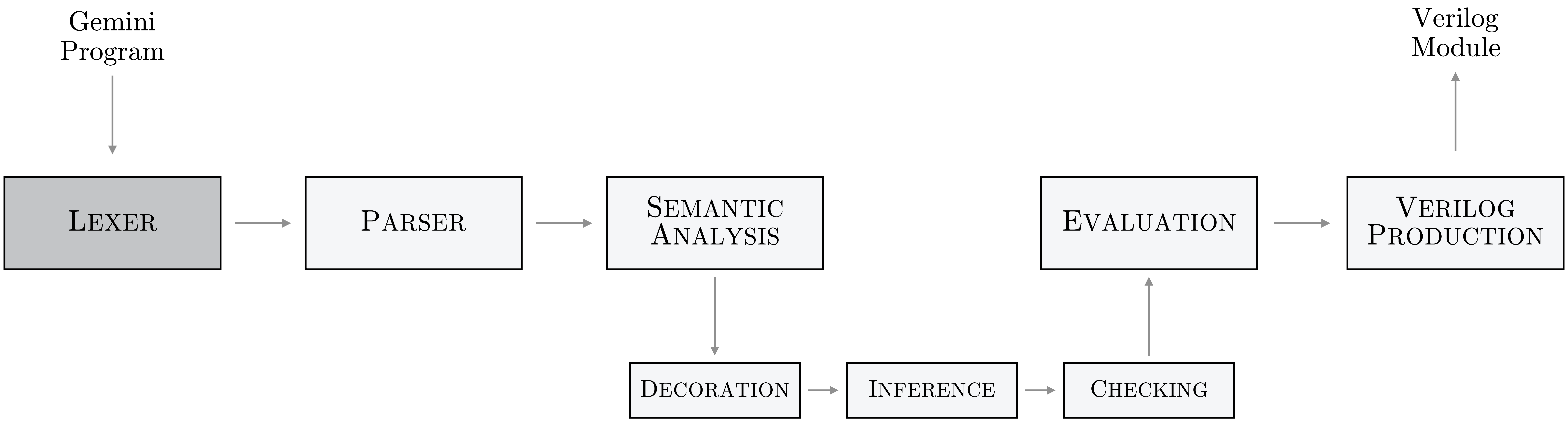

1
Motivation
2
Preliminary Research
3
Language Specification
4
Design
5
Implementation
6
Demonstration
7
Future Work
8
Conclusion

Implementation

Implementation



Implementation



Implementation // **Lexer**

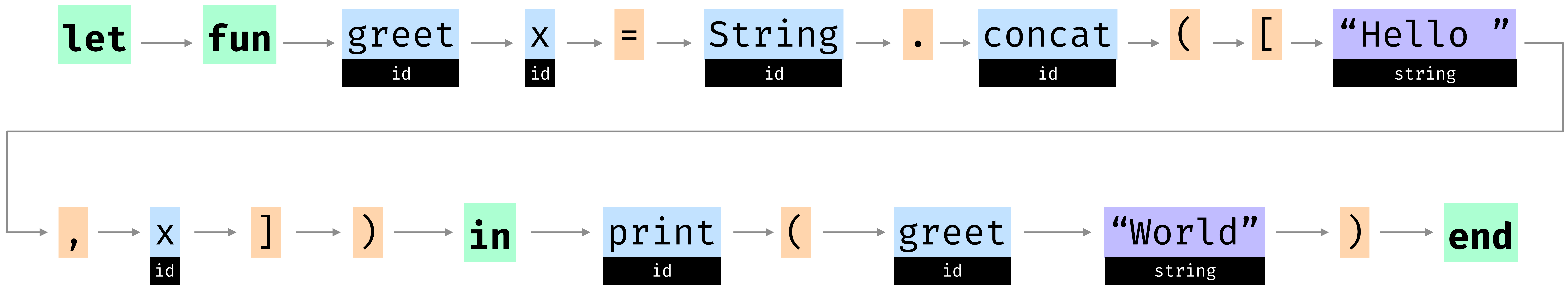
Scans program and produces lexemes, classified into token classes

Scans program and produces lexemes, classified into token classes

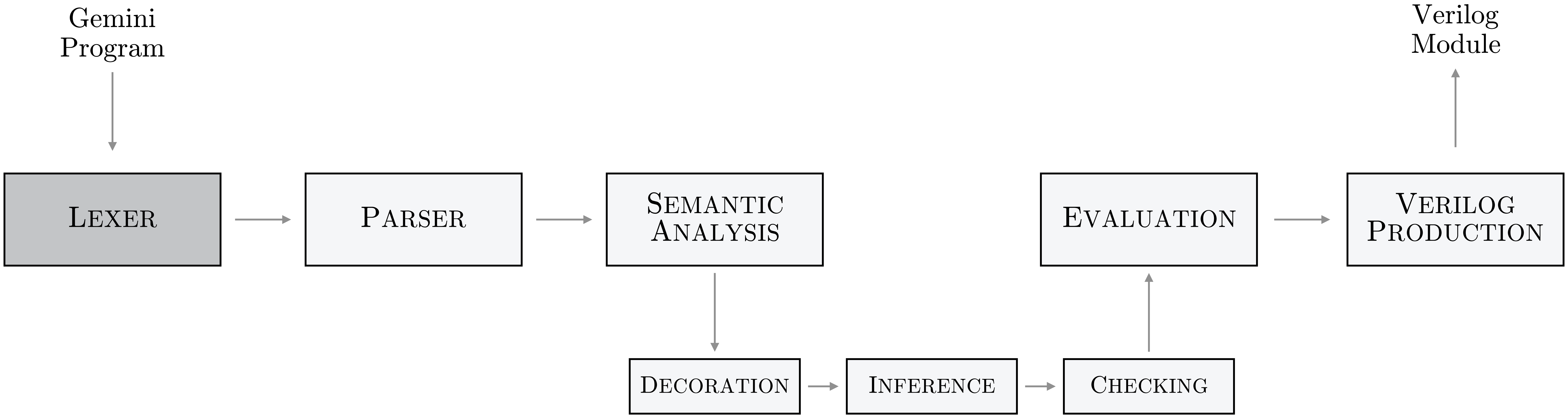
```
let
    fun greet x = String.concat([“Hello ”,  x])
in
    print(greet “World”)
end
```

Implementation // Lexer

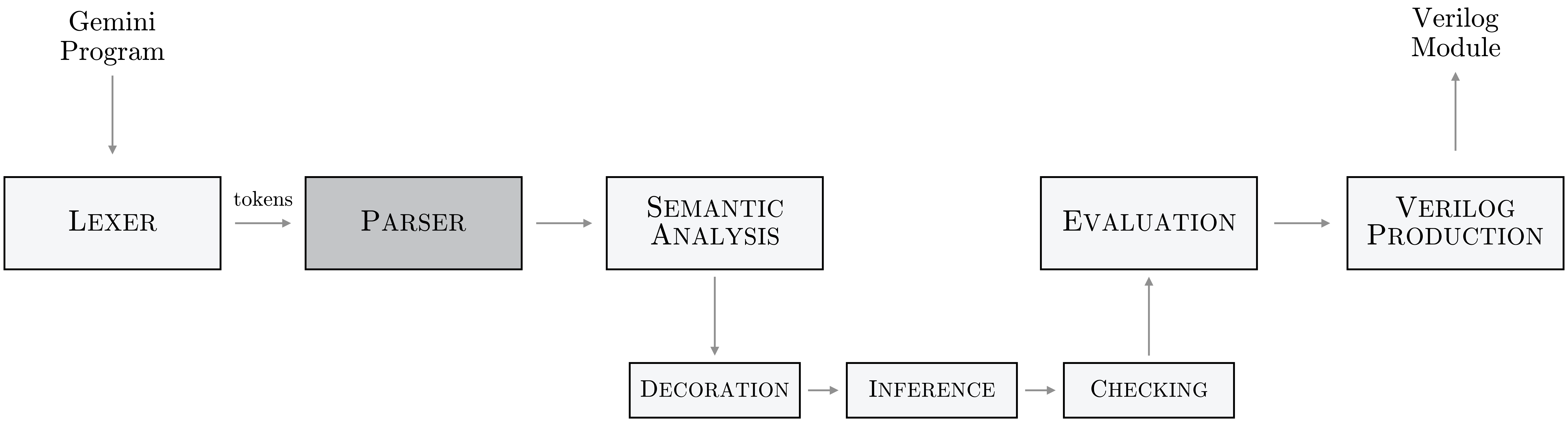
Scans program and produces lexemes, classified into token classes



Implementation



Implementation



Implementation // Parser

At this point, we know program is syntactically correct

Implementation // Parser

At this point, we know program is syntactically correct

Problem

It may not be grammatically correct (Ex: `42 +`)

At this point, we know program is syntactically correct

Problem

It may not be grammatically correct (Ex: `42 +`)

Solution

We must verify the grammar

At this point, we know program is syntactically correct

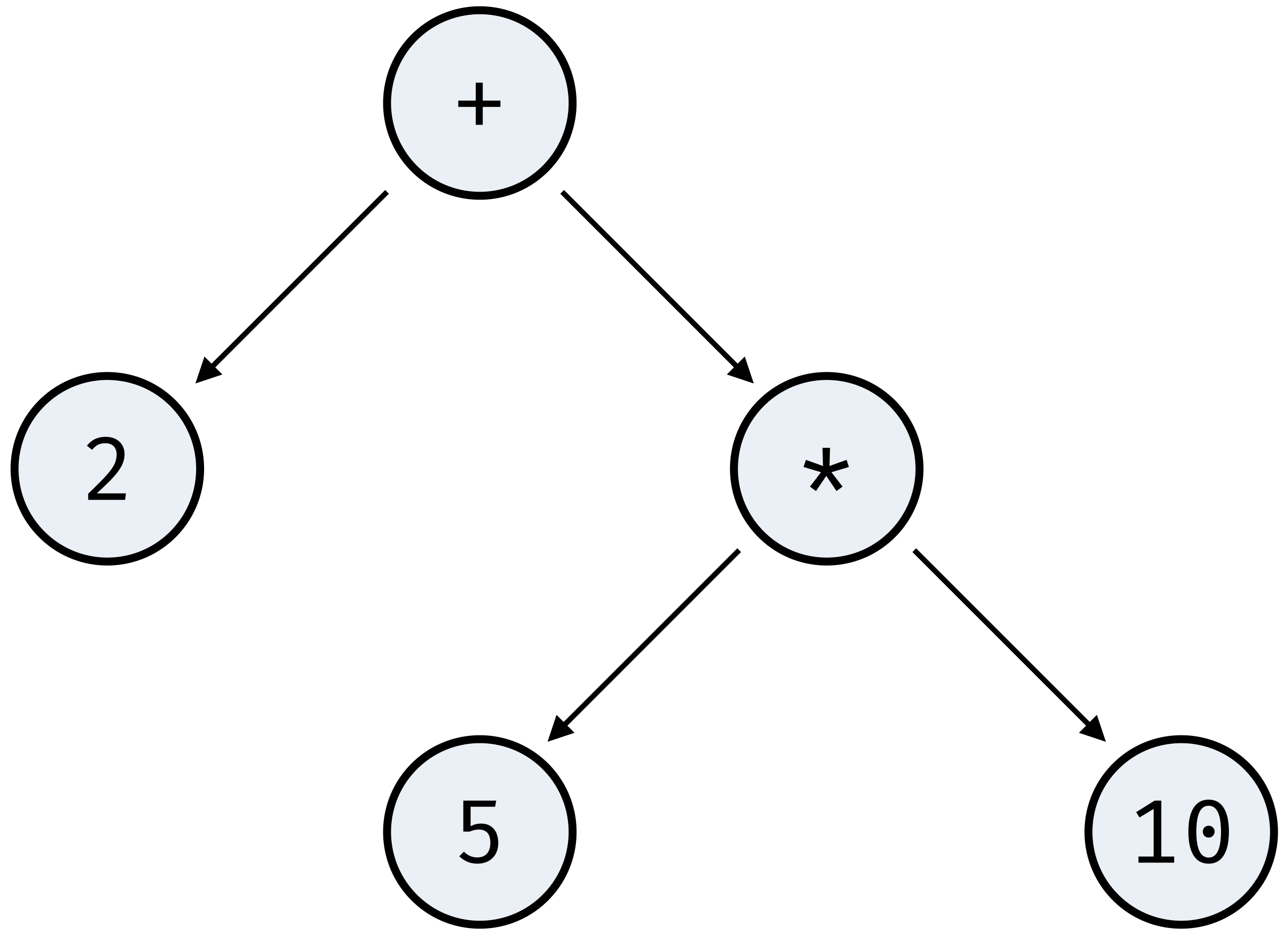
Problem

It may not be grammatically correct (Ex: `42 +`)

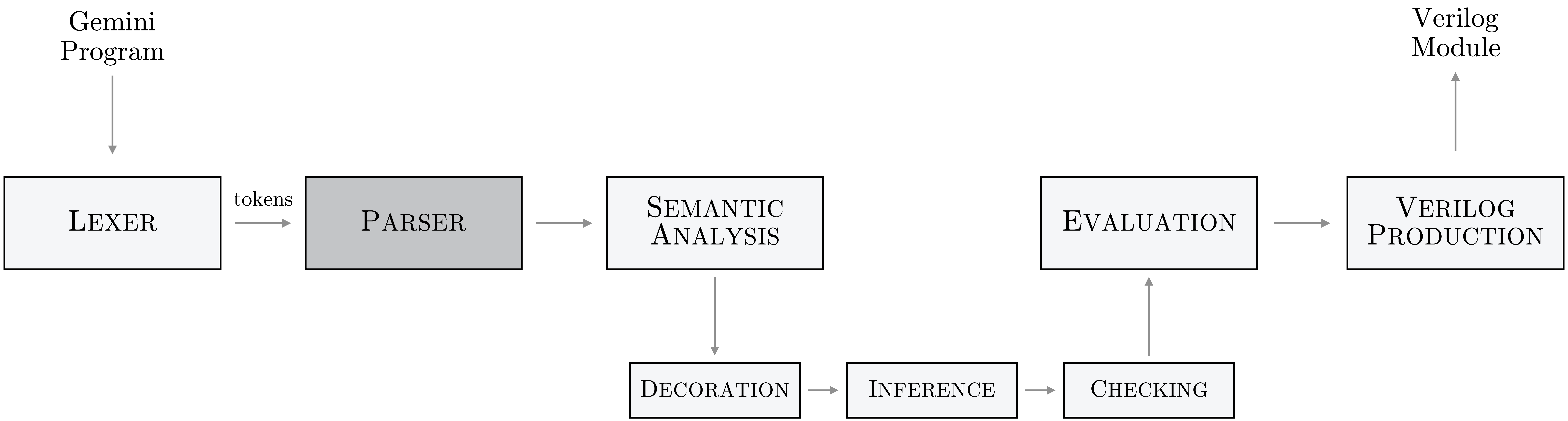
Solution

We must verify the grammar

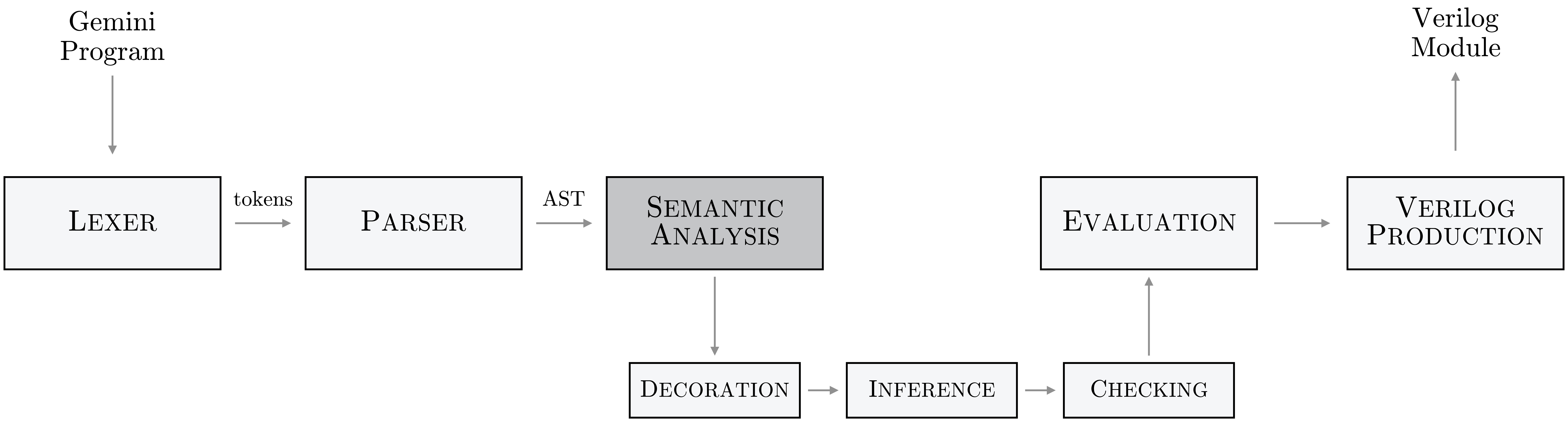
`2 + 5 * 10`



IMPLEMENTATION



Implementation



At this point, we know program is syntactically and grammatically correct

At this point, we know program is syntactically and grammatically correct

Problem

It may not be semantically correct (Ex: `“hello” + 42`)

At this point, we know program is syntactically and grammatically correct

Problem

It may not be semantically correct (Ex: `“hello” + 42`)

Solution

We must verify the type semantics

Preventative Issue

Not all types are currently known since variables may be written with implicit types

Preventative Issue

Not all types are currently known since variables may be written with implicit types

```
fun add(x, y) = x + y
```

Preventative Issue

Not all types are currently known since variables may be written with implicit types

```
fun add(x, y) = x + y
```

It is the responsibility of the compiler to infer the actual types

Semantic Analysis

1. Decoration
2. Inference
3. Checking

1. Decoration
2. Inference
3. Checking

We translate from Gemini to a fictional intermediate language called ExplicitGemini

We translate from Gemini to a fictional intermediate language called ExplicitGemini

```
fun print_and_mult(x    , y    , s : string)     = (print(s); x * y)
```

We translate from Gemini to a fictional intermediate language called ExplicitGemini

```
fun print_and_mult(x :'a, y :'b, s : string) :'c = (print(s); x * y)
```

We translate from Gemini to a fictional intermediate language called ExplicitGemini

```
fun print_and_mult(x :'a, y :'b, s : string) :'c = (print(s); x * y)
```

`'a` represents a type variable, which we try to infer based on how it is used

1. Decoration
2. Inference
3. Checking

With all variables decorated, we infer all types as generally as possible

With all variables decorated, we infer all types as generally as possible

Two underlying algorithms

1. Unification
2. Substitution

Algorithm: Unification computes the smallest possible substitution mapping from type variables to types

```
fun doubleList (mylist : ‘a) : ‘b = case mylist of
                                      [] : ‘c ⇒ [] : ‘d
                                    |: (x : ‘e)::(rest : ‘f) ⇒ (x * 2)::(doubleList rest) : ‘g
```

Algorithm: Unification computes the smallest possible substitution mapping from type variables to types

```
fun doubleList (mylist : 'a) : 'b = case mylist of
                                        [] : 'c ⇒ [] : 'd
                                      |: (x : 'e)::(rest : 'f) ⇒ (x * 2)::(doubleList rest) : 'g
```

```
{'c ↦ 'h list}
```

Algorithm: Unification computes the smallest possible substitution mapping from type variables to types

```
fun doubleList (mylist : ‘a) : ‘b = case mylist of
                                        [] : ‘c ⇒ [] : ‘d
                                      |: (x : ‘e)::(rest : ‘f) ⇒ (x * 2)::(doubleList rest) : ‘g
```

```
{‘c ↦ ‘h list}

{‘d ↦ ‘i list}
```

Algorithm: Unification computes the smallest possible substitution mapping from type variables to types

```
fun doubleList (mylist : ‘a) : ‘b = case mylist of
                                       [] : ‘c ⇒ [] : ‘d
                                    |: (x : ‘e)::(rest : ‘f) ⇒ (x * 2)::(doubleList rest) : ‘g
```

```
{‘c ↦ ‘h list}

{‘d ↦ ‘i list}

{‘e ↦ int}
```

Algorithm: Unification computes the smallest possible substitution mapping from type variables to types

```
fun doubleList (mylist : ‘a) : ‘b = case mylist of
                                          [] : ‘c ⇒ [] : ‘d
                                        |: (x : ‘e)::(rest : ‘f) ⇒ (x * 2)::(doubleList rest) : ‘g
```

```
{‘c ↦ ‘h list}

{‘d ↦ ‘i list}

{‘e ↦ int}

{‘f ↦ ‘e list}
```

Algorithm: Unification computes the smallest possible substitution mapping from type variables to types

```
fun doubleList (mylist : ‘a) : ‘b = case mylist of
                                        [] : ‘c ⇒ [] : ‘d
                                      |: (x : ‘e)::(rest : ‘f) ⇒ (x * 2)::(doubleList rest) : ‘g
```

```
{‘c ↦ ‘h list}

{‘d ↦ ‘i list}

{‘e ↦ int}

{‘f ↦ int list}
```

Algorithm: <u>Unification</u> computes the smallest possible substitution mapping from type variables to types

```
fun doubleList (mylist : ‘a) : ‘b = case mylist of
                                        [] : ‘c ⇒ [] : ‘d
                                      |: (x : ‘e)::(rest : ‘f) ⇒ (x * 2)::(doubleList rest) : ‘g
```

```
{‘c ↦ ‘h list}          {‘g ↦ ‘e list}

{‘d ↦ ‘i list}

{‘e ↦ int}

{‘f ↦ int list}
```

Algorithm: Unification computes the smallest possible substitution mapping from type variables to types

```
fun doubleList (mylist : ‘a) : ‘b = case mylist of
                                        [] : ‘c ⇒ [] : ‘d
                                       |: (x : ‘e)::(rest : ‘f) ⇒ (x * 2)::(doubleList rest) : ‘g
```

```
{‘c ↦ ‘h list}            {‘g ↦ int list}

{‘d ↦ ‘i list}

{‘e ↦ int}

{‘f ↦ int list}
```

Algorithm: Unification computes the smallest possible substitution mapping from type variables to types

```
fun doubleList (mylist : ‘a) : ‘b = case mylist of
                                        [] : ‘c ⇒ [] : ‘d
                                      |: (x : ‘e)::(rest : ‘f) ⇒ (x * 2)::(doubleList rest) : ‘g
```

```
{‘c ↦ ‘h list}           {‘g ↦ int list}

{‘d ↦ ‘g}

{‘e ↦ int}

{‘f ↦ int list}
```

Algorithm: Unification computes the smallest possible substitution mapping from type variables to types

```
fun doubleList (mylist : ‘a) : ‘b = case mylist of
                                          [] : ‘c ⇒ [] : ‘d
                                        |: (x : ‘e)::(rest : ‘f) ⇒ (x * 2)::(doubleList rest) : ‘g
```

```
{‘c ↦ ‘h list}          {‘g ↦ int list}

{‘d ↦ int list}

{‘e ↦ int}

{‘f ↦ int list}
```

Algorithm: Unification computes the smallest possible substitution mapping from type variables to types

```
fun doubleList (mylist : ‘a) : ‘b = case mylist of
                                         [] : ‘c ⇒ [] : ‘d
                                       |: (x : ‘e)::(rest : ‘f) ⇒ (x * 2)::(doubleList rest) : ‘g
```

```
{‘c ↦ int list}          {‘g ↦ int list}

{‘d ↦ int list}

{‘e ↦ int}

{‘f ↦ int list}
```

Algorithm: Unification computes the smallest possible substitution mapping from type variables to types

```
fun doubleList (mylist : ‘a) : ‘b = case mylist of
                                          [] : ‘c ⇒ [] : ‘d
                                        |: (x : ‘e)::(rest : ‘f) ⇒ (x * 2)::(doubleList rest) : ‘g
```

```
{‘c ↦ int list}        {‘g ↦ int list}

{‘d ↦ int list}        {‘a ↦ ‘c}

{‘e ↦ int}

{‘f ↦ int list}
```

Algorithm: <u>Unification</u> computes the smallest possible substitution mapping from type variables to types

```
fun doubleList (mylist : ‘a) : ‘b = case mylist of
                                         [] : ‘c ⇒ [] : ‘d
                                       |: (x : ‘e)::(rest : ‘f) ⇒ (x * 2)::(doubleList rest) : ‘g
```

```
{‘c ↦ int list}          {‘g ↦ int list}

{‘d ↦ int list}          {‘a ↦ int list}

{‘e ↦ int}

{‘f ↦ int list}
```

Algorithm: Unification computes the smallest possible substitution mapping from type variables to types

```
fun doubleList (mylist : ‘a) : ‘b = case mylist of
                                       [] : ‘c ⇒ [] : ‘d
                                     |: (x : ‘e)::(rest : ‘f) ⇒ (x * 2)::(doubleList rest) : ‘g
```

```
{‘c ↦ int list}          {‘g ↦ int list}
{‘d ↦ int list}          {‘a ↦ int list}
{‘e ↦ int}               {‘b ↦ ‘d}
{‘f ↦ int list}
```

Algorithm: Unification computes the smallest possible substitution mapping from type variables to types

```
fun doubleList (mylist : ‘a) : ‘b = case mylist of
                                        [] : ‘c ⇒ [] : ‘d
                                      |: (x : ‘e)::(rest : ‘f) ⇒ (x * 2)::(doubleList rest) : ‘g
```

```
{‘c ↦ int list}         {‘g ↦ int list}
{‘d ↦ int list}         {‘a ↦ int list}
{‘e ↦ int}              {‘b ↦ int list}
{‘f ↦ int list}
```

Algorithm: Unification computes the smallest possible substitution mapping from type variables to types

```
fun doubleList (mylist : ‘a) : ‘b = case mylist of
                                        [] : ‘c ⇒ [] : ‘d
                                      |: (x : ‘e)::(rest : ‘f) ⇒ (x * 2)::(doubleList rest) : ‘g
```

```
{‘c ↦ int list}        {‘g ↦ int list}        doubleList : int list → int list
{‘d ↦ int list}        {‘a ↦ int list}
{‘e ↦ int}             {‘b ↦ int list}
{‘f ↦ int list}
```

Algorithm: Substitution applies the mappings to the variable and type environments

In some cases, type variables do not get substituted

In some cases, type variables do not get substituted

```
fun concat e mylist = e::mylist
```

In some cases, type variables do not get substituted

```
fun concat e mylist = e::mylist
```

The type of `concat` is `‘a → ‘a list → ‘a list`

In some cases, type variables do not get substituted

```
fun concat e mylist = e::mylist
```

The type of `concat` is `‘a → ‘a list → ‘a list`

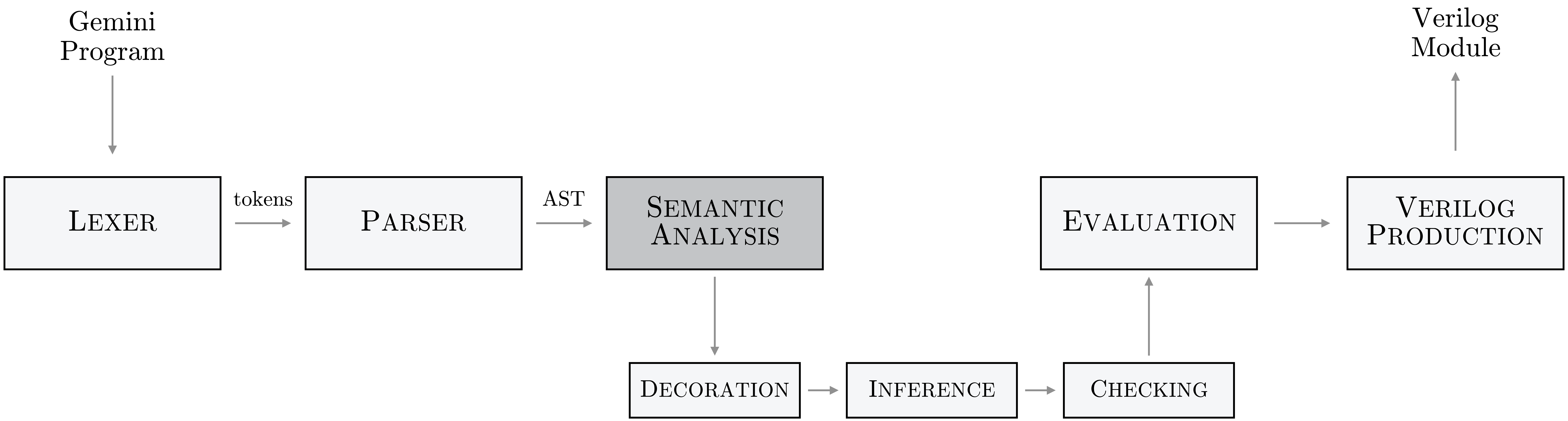
In type theoretical terms, $\forall \mathrm{a}.\lambda \mathrm{x} : \mathrm{a}.\lambda \mathrm{y} : \mathrm{a}\ \mathrm{list}.\mathrm{x}::\mathrm{y}$

Implementation // Semantic Analysis

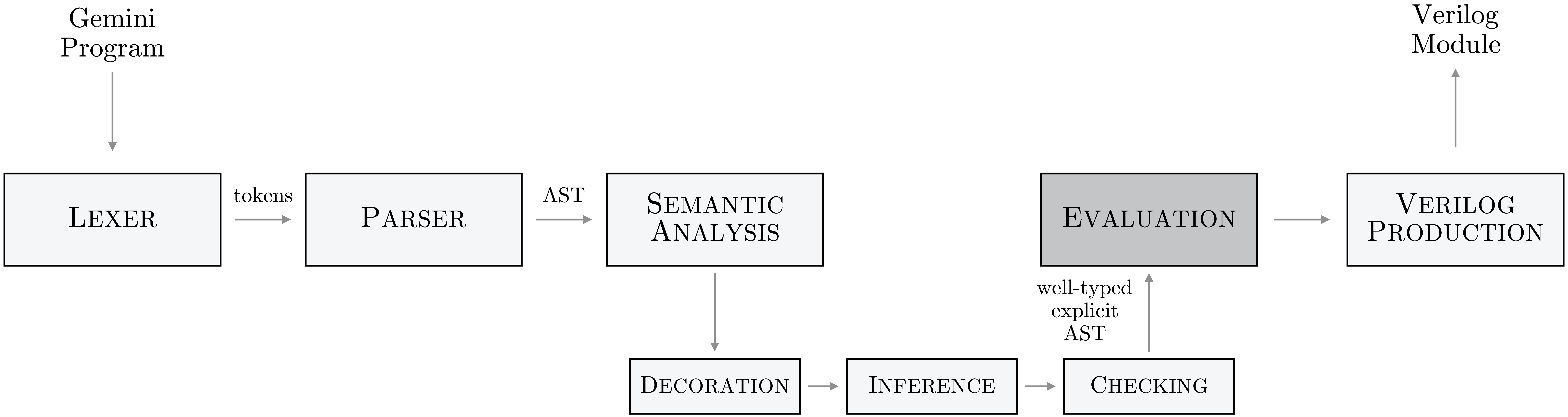
1. Decoration
2. Inference
3. Checking

Now that all types are inferred, we can enforce typing rules that we formalized earlier

IMPLEMENTATION



Implementation



At this point, we have a well-typed AST consisting of software and hardware expressions

At this point, we have a well-typed AST consisting of software and hardware expressions

Problem

Verilog only supports hardware values

At this point, we have a well-typed AST consisting of software and hardware expressions

Problem
Verilog only supports hardware values

Solution
Execute all software expressions to generate a hardware-only tree

Implementation // Evaluation

Evaluation is similar to algorithms in the past

Evaluation

Evaluation is similar to algorithms in the past

1. We recurse over the AST and evaluate each expression, which may contain subexpressions

Implementation // Evaluation

Evaluation is similar to algorithms in the past

1. We recurse over the AST and evaluate each expression, which may contain subexpressions
2. Base case of recursion is when a variable or constant is encountered

Implementation // Evaluation

Evaluation is similar to algorithms in the past

1. We recurse over the AST and evaluate each expression, which may contain subexpressions
2. Base case of recursion is when a variable or constant is encountered
3. At each node, subexpressions are evaluated and used to evaluate the node itself

Evaluation is similar to algorithms in the past

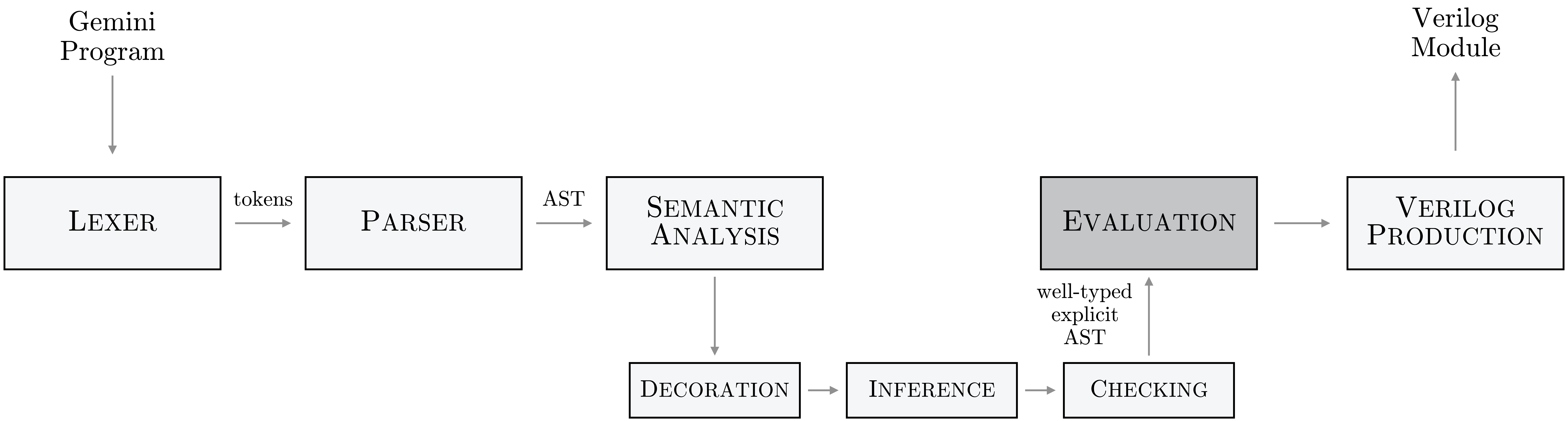
1. We recurse over the AST and evaluate each expression, which may contain subexpressions

2. Base case of recursion is when a variable or constant is encountered

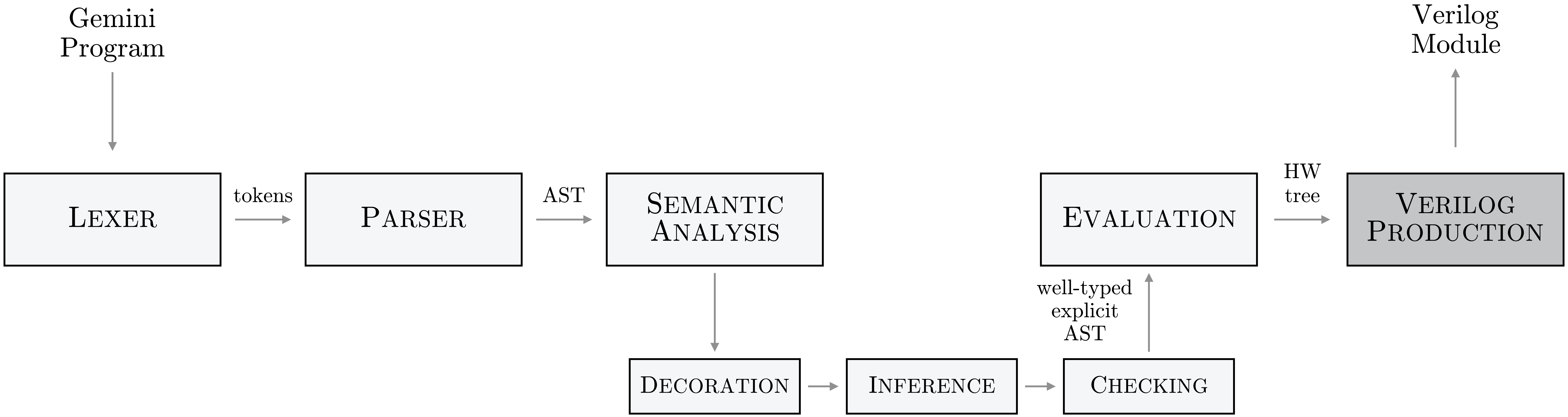
3. At each node, subexpressions are evaluated and used to evaluate the node itself

4. For each declaration, the value store is augmented to map from the variable name to its value

Implementation



Implementation



Implementation // Verilog Production

At this point, we have a tree of hardware values representing a circuit consisting only of bits, gates, arrays, records, and pins (variables)

At this point, we have a tree of hardware values representing a circuit consisting only of bits, gates, arrays, records, and pins (variables)

Problem
The structure represents a module but is not in an executable format

Implementation // Verilog Production

At this point, we have a tree of hardware values representing a circuit consisting only of bits, gates, arrays, records, and pins (variables)

Problem
The structure represents a module but is not in an executable format

Solution
We must output Verilog that represents the tree

Verilog Production

Verilog production is similar to algorithms in the past

1. We recurse over the AST and produce Verilog for each expression, which may contain subexpressions

Implementation // Verilog Production

Verilog production is similar to algorithms in the past

1. We recurse over the AST and produce Verilog for each expression, which may contain subexpressions

2. Base case of recursion is when a pin or constant is encountered

Implementation // Verilog Production

Verilog production is similar to algorithms in the past

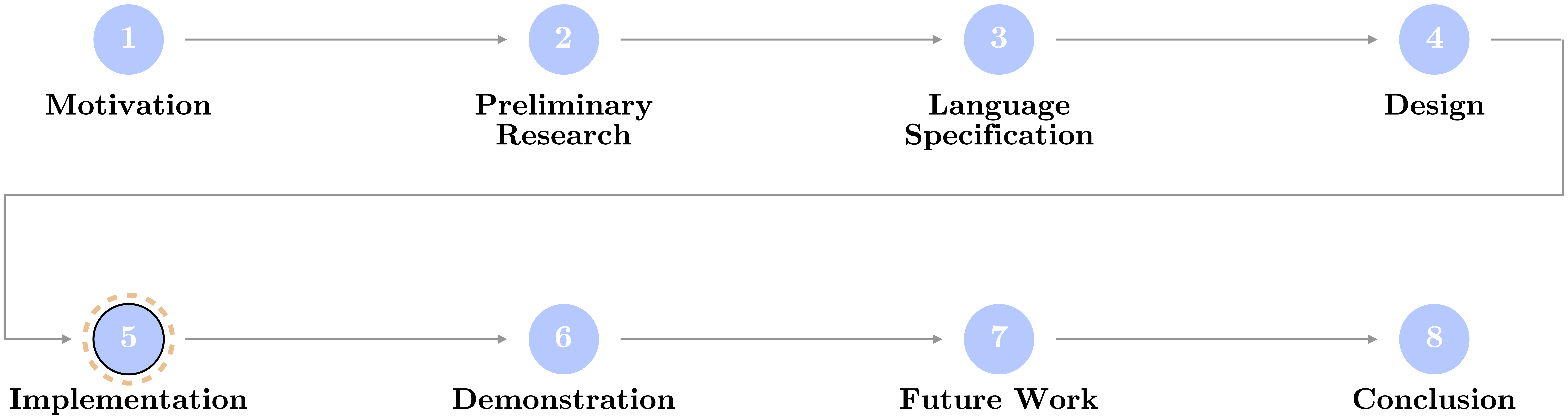
1. We recurse over the AST and produce Verilog for each expression, which may contain subexpressions
2. Base case of recursion is when a pin or constant is encountered
3. At each node, subexpressions are evaluated and used to evaluate the node itself

Implementation // Verilog Production

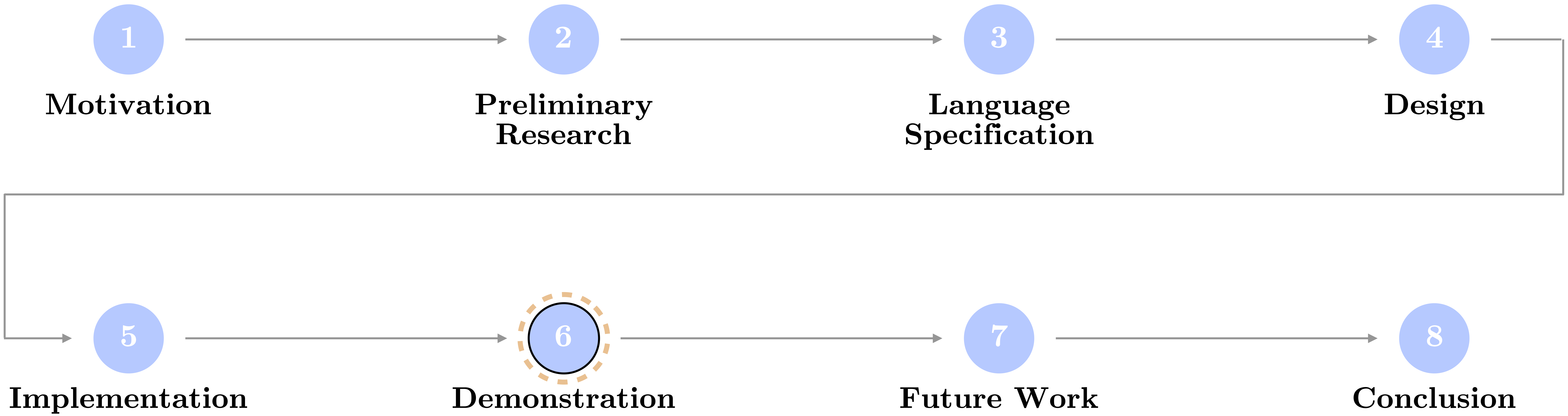
Verilog production is similar to algorithms in the past

1. We recurse over the AST and produce Verilog for each expression, which may contain subexpressions
2. Base case of recursion is when a pin or constant is encountered
3. At each node, subexpressions are evaluated and used to evaluate the node itself
4. For each node, fresh wire is generated and returned for superexpressions to use

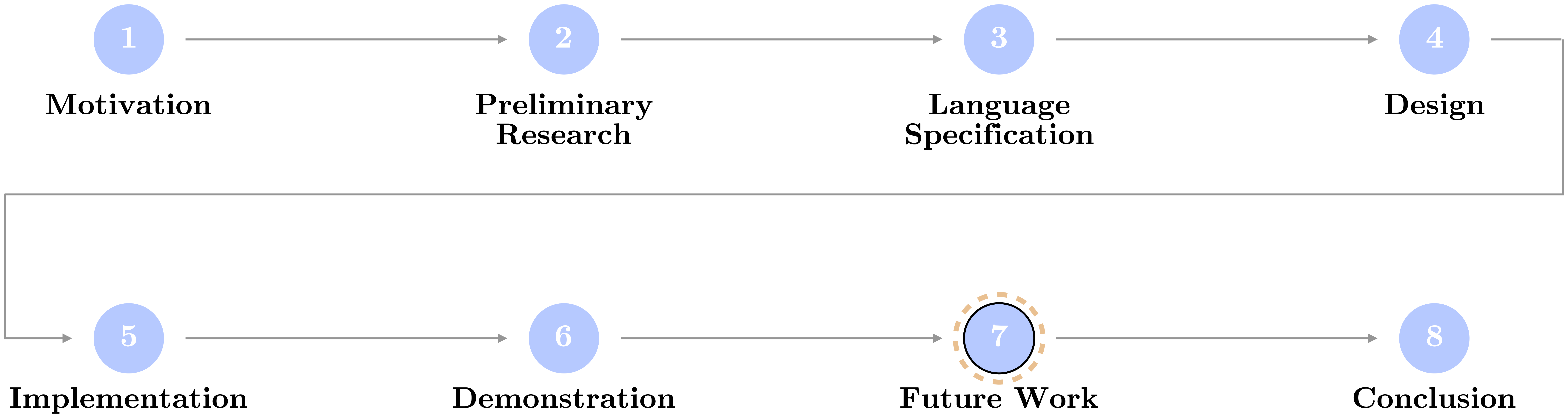
Roadmap


1
Motivation
2
Preliminary Research
3
Language Specification
4
Design
5
Implementation
6
Demonstration
7
Future Work
8
Conclusion

Roadmap


1
Motivation
2
Preliminary Research
3
Language Specification
4
Design
5
Implementation
6
Demonstration
7
Future Work
8
Conclusion

Roadmap


1
Motivation
2
Preliminary Research
3
Language Specification
4
Design
5
Implementation
6
Demonstration
7
Future Work
8
Conclusion

Future Work

Extensions

Optimizations

Future Work

Extensions

- Simulation/waveform backend
- Testbench generation

Optimizations

Future Work

Extensions

- Simulation/waveform backend
- Testbench generation

Optimizations

- Dataflow analysis to reuse wires
- Constant propagation
- Constraining resources finitely

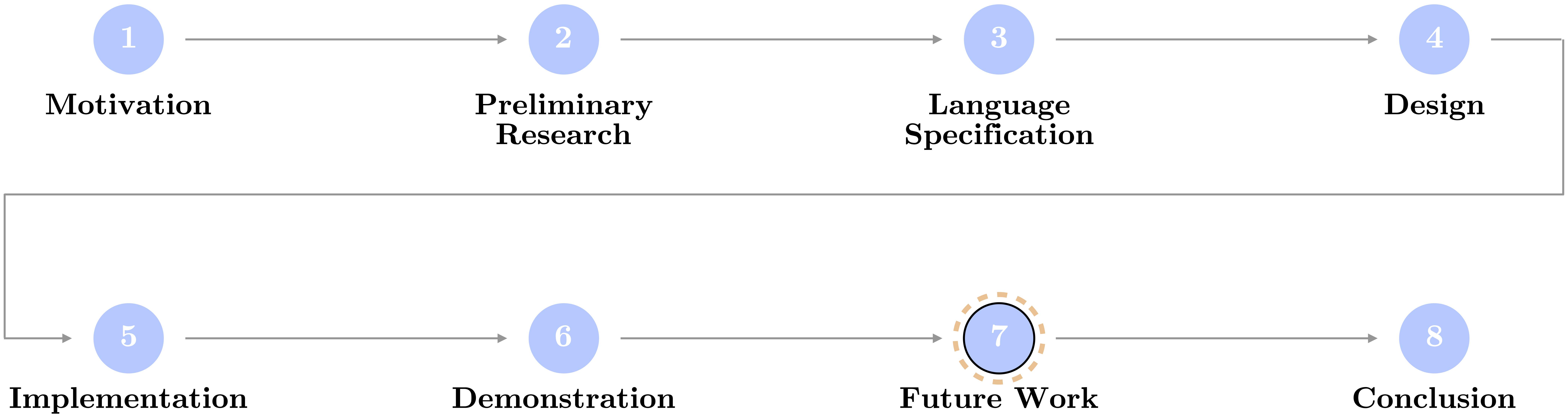
Future Work

Plan to Open-Source

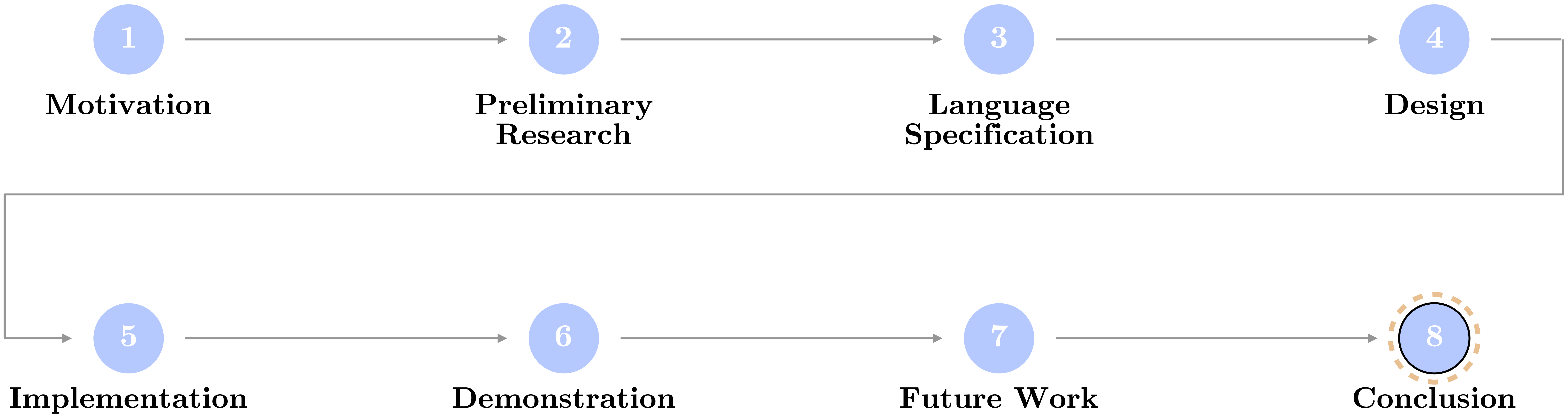
All code is hosted on GitHub

Extensive documentation at bit.ly/gemini-docs

Roadmap


1
Motivation
2
Preliminary Research
3
Language Specification
4
Design
5
Implementation
6
Demonstration
7
Future Work
8
Conclusion

Roadmap


1
Motivation
2
Preliminary Research
3
Language Specification
4
Design
5
Implementation
6
Demonstration
7
Future Work
8
Conclusion

Conclusion

I spent the last year answering the following questions:

Question 1
Can I design a programming language that combines the powerful features of software programming languages with the ability to describe electronic circuits?

Question 2
Can I develop a compiler that accepts a program in this language and produces an optimized Verilog module?

Conclusion

I spent the last year answering the following questions:

✓ **Question 1**
Can I design a programming language that combines the powerful features of software programming languages with the ability to describe electronic circuits?

✓ **Question 2**
Can I develop a compiler that accepts a program in this language and produces an optimized Verilog module?

Conclusion

I spent the last year answering the following questions:

✓ **Question 1**
Can I design a programming language that combines the powerful features of software programming languages with the ability to describe electronic circuits?

✓ **Question 2**
Can I develop a compiler that accepts a program in this language and produces an optimized Verilog module?

Answer
Yes, and Gemini is proof that unconventional features like multiple kinds, value-parameterized types, and the manifestation of time are possible to implement.

References

[1] Barbacci, M. ”A comparison of register transfer languages for describing computers and digital systems,” Carnegie-Mellon Univ., Dept. of Computer Science, March 1973

[2] ”Verilog's inventor nabs EDA's Kaufman award”. EE Times. November 7, 2005

[3] Department of Defense (1992). Military Standard, Standard general requirements for electronic equipment. Retrieved November 15, 2017

[4] Barbacci, M., Grout S., Lindstrom, G., Maloney, M.P. ”Ada as a hardware description language : an initial report,” Carnegie-Mellon Univ., Dept. of Computer Science, 1984

[5] Pierce, Benjamin C. Types and Programming Languages. MIT Press, 2002

[6] Appel, Andrew W., et al. “A Lexical Analyzer Generator for Standard ML.” A Lexical Analyzer Generator for Standard ML. Version 1.6.0, October 1994, Princeton University, Oct. 1994, www.smlnj.org/doc/ML- Lex/ manual.html.

[7] Tarditi, David R., and Andrew W. Appel. “ML-Yacc User's Manual Version 2.3.” Princeton University, The Trustees of Princeton University, 6 Oct. 1994, www.cs.princeton.edu/˜appel/modern/ml/ml-yacc/manual.html

Questions?