Let's Write a Parser

Ionuț G. Stan — I T.A.K.E. — May 2016

About Me

About Me

- Software Developer at Eloquentix

About Me

- Software Developer at Eloquentix
- I work mostly with Scala

About Me

- Software Developer at Eloquentix
- I work mostly with Scala
- I like FP, programming languages, compilers

About Me

- Software Developer at Eloquentix
- I work mostly with Scala
- I like FP, programming languages, compilers
- I started the Bucharest FP meet-up group

About Me

- Software Developer at Eloquentix
- I work mostly with Scala
- I like FP, programming languages, compilers
- I started the Bucharest FP meet-up group
- I occasionally blog on igstan.ro

Plan

Plan

- Vehicle Language: µML

Plan

- Vehicle Language: μML
- Compilers Overview

Plan

- Vehicle Language: μML
- Compilers Overview
- Parsing: Intuitions and Live Coding

Vehicle Language: μML

Vehicle Language: µML

1. Integers: `1`, `23`, `456`, etc.

Vehicle Language: μML

1. Integers: `1`, `23`, `456`, etc.
2. Identifiers (only letters): `inc`, `cond`, `a`, etc.

Vehicle Language: μML

1. Integers: `1`, `23`, `456`, etc.
2. Identifiers (only letters): `inc`, `cond`, `a`, etc.
3. Booleans: `true` and `false`

Vehicle Language: μML

1. Integers: `1`, `23`, `456`, etc.
2. Identifiers (only letters): `inc`, `cond`, `a`, etc.
3. Booleans: `true` and `false`
4. Single-argument anonymous functions: `fn a => a`

Vehicle Language: μML

1. Integers: `1`, `23`, `456`, etc.
2. Identifiers (only letters): `inc`, `cond`, `a`, etc.
3. Booleans: `true` and `false`
4. Single-argument anonymous functions: `fn a => a`
5. Function application: `inc 42`

Vehicle Language: µML

1. Integers: `1`, `23`, `456`, etc.
2. Identifiers (only letters): `inc`, `cond`, `a`, etc.
3. Booleans: `true` and `false`
4. Single-argument anonymous functions: `fn a => a`
5. Function application: `inc 42`
6. If expressions: `if cond then t else f`

Vehicle Language: μML

1. Integers: `1`, `23`, `456`, etc.
2. Identifiers (only letters): `inc`, `cond`, `a`, etc.
3. Booleans: `true` and `false`
4. Single-argument anonymous functions: `fn a => a`
5. Function application: `inc 42`
6. If expressions: `if cond then t else f`
7. Addition and subtraction: `a + b`, `a - b`

Vehicle Language: µML

1. Integers: `1`, `23`, `456`, etc.
2. Identifiers (only letters): `inc`, `cond`, `a`, etc.
3. Booleans: `true` and `false`
4. Single-argument anonymous functions: `fn a => a`
5. Function application: `inc 42`
6. If expressions: `if cond then t else f`
7. Addition and subtraction: `a + b`, `a - b`
8. Parenthesized expressions: `(a + b)`

Vehicle Language: μML

9. Let blocks/expressions:

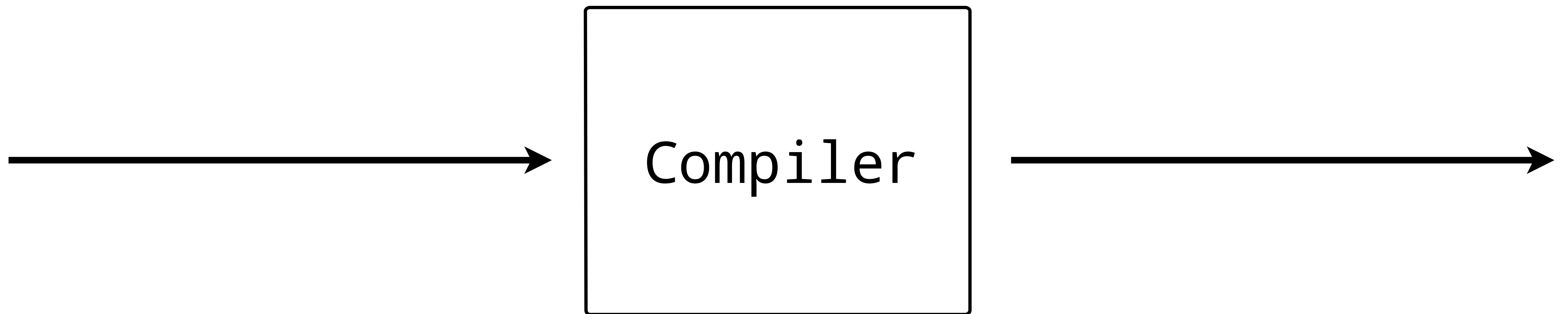
```
let
  val name = ...
in
  name
end
```

Small Example

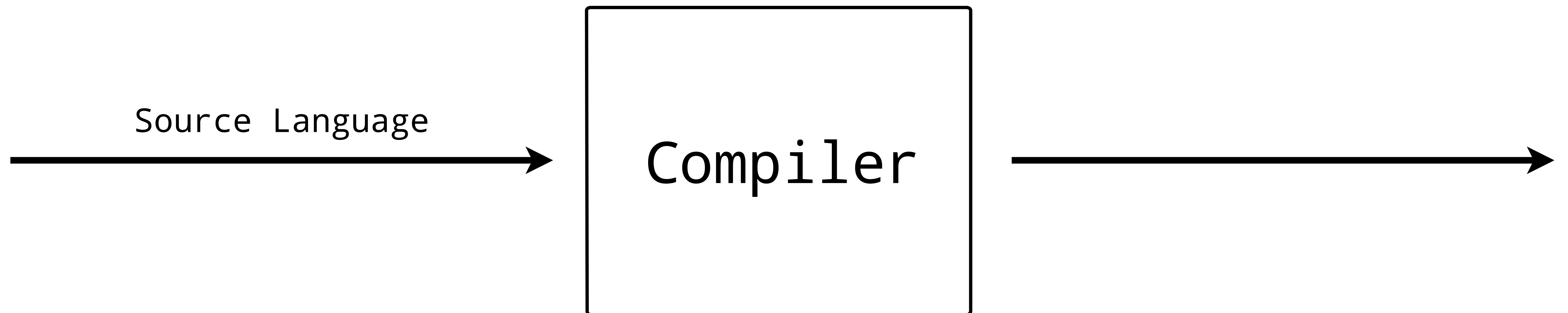
```
let
  val inc =
    fn a => a + 1
in
  inc 42
end
```

Compilers Overview

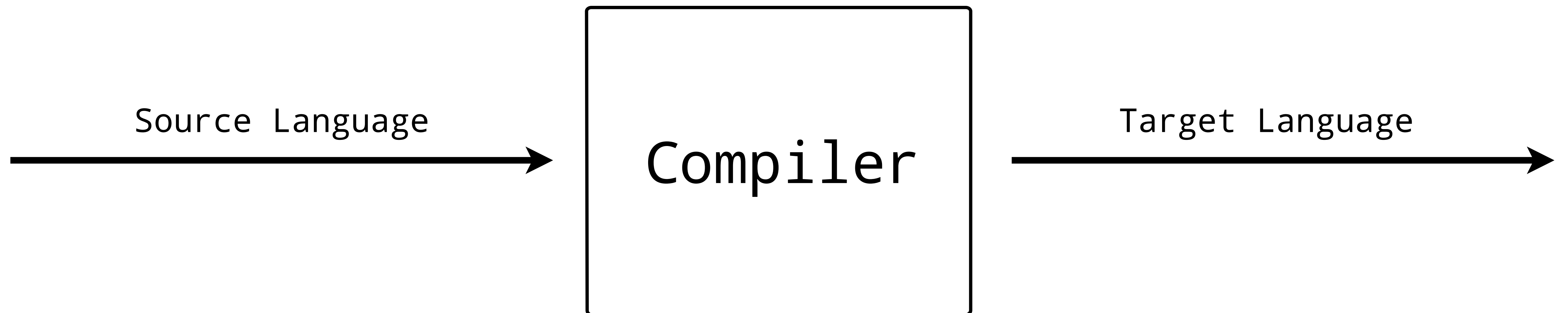
Compilers Overview



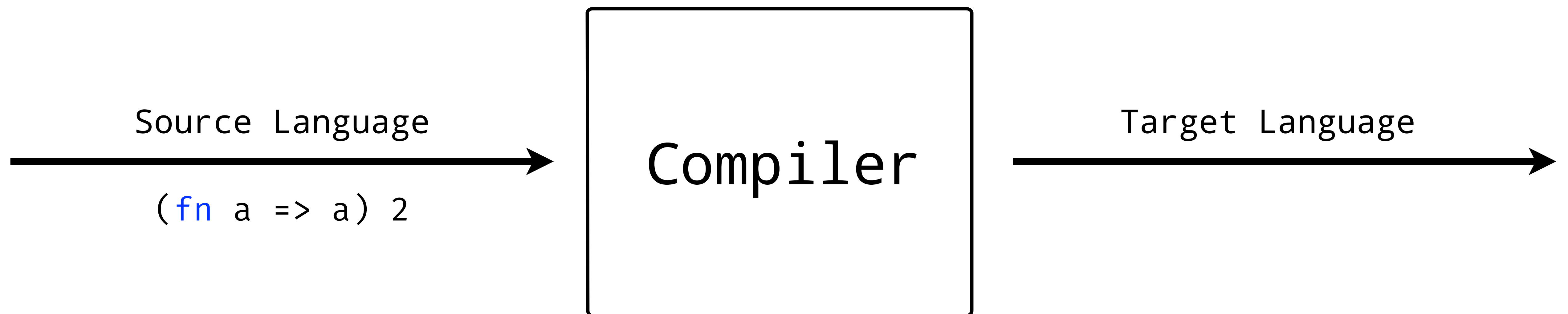
Compilers Overview



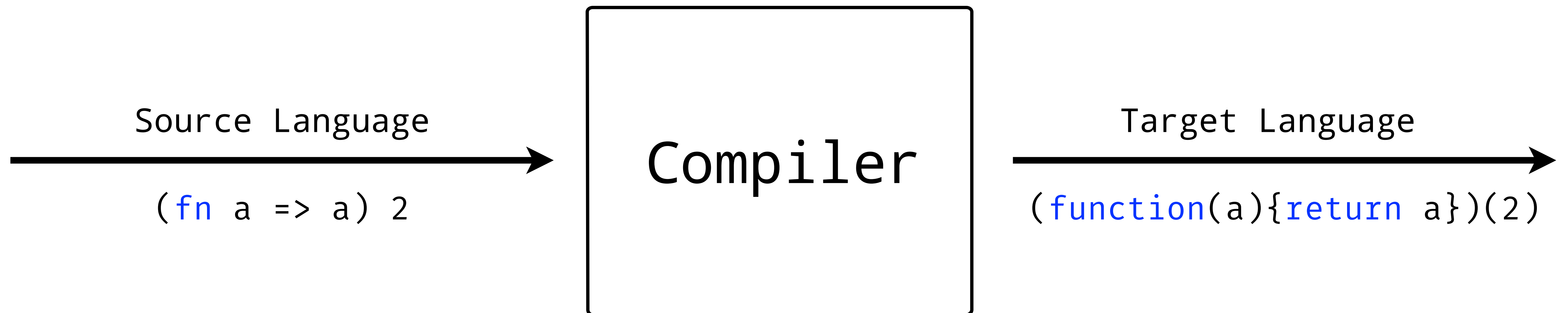
Compilers Overview



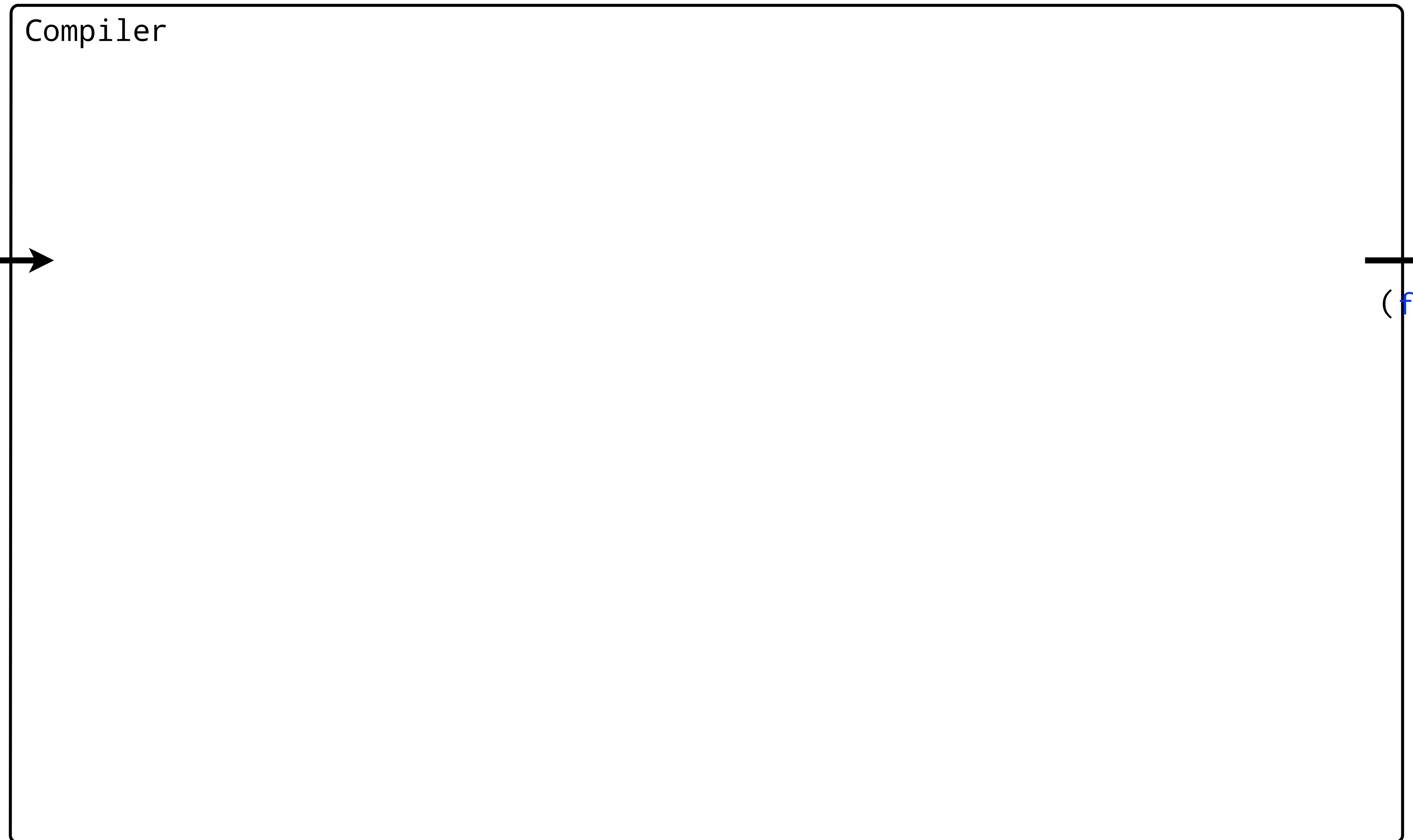
Compilers Overview



Compilers Overview

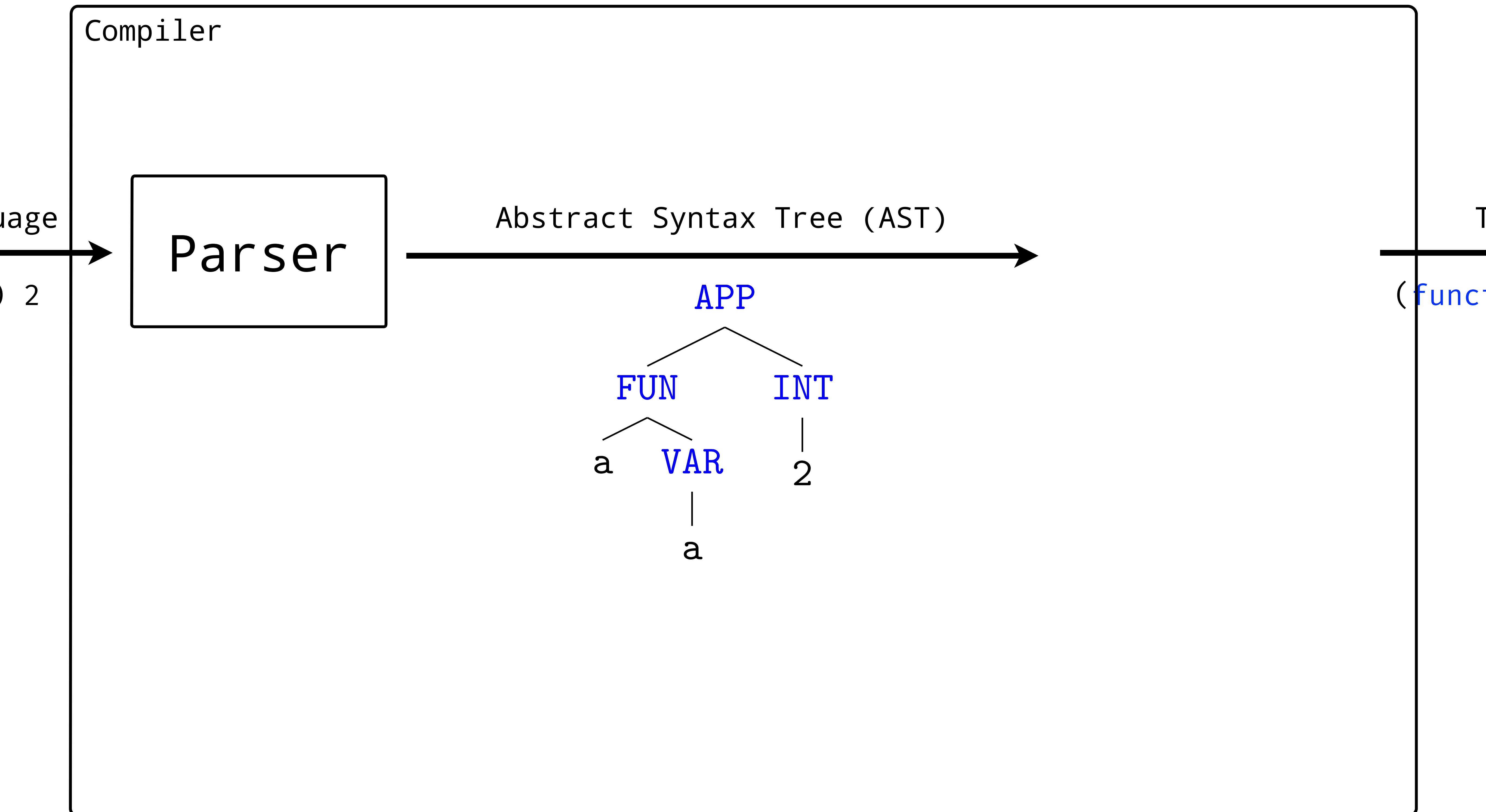


Compilers Overview

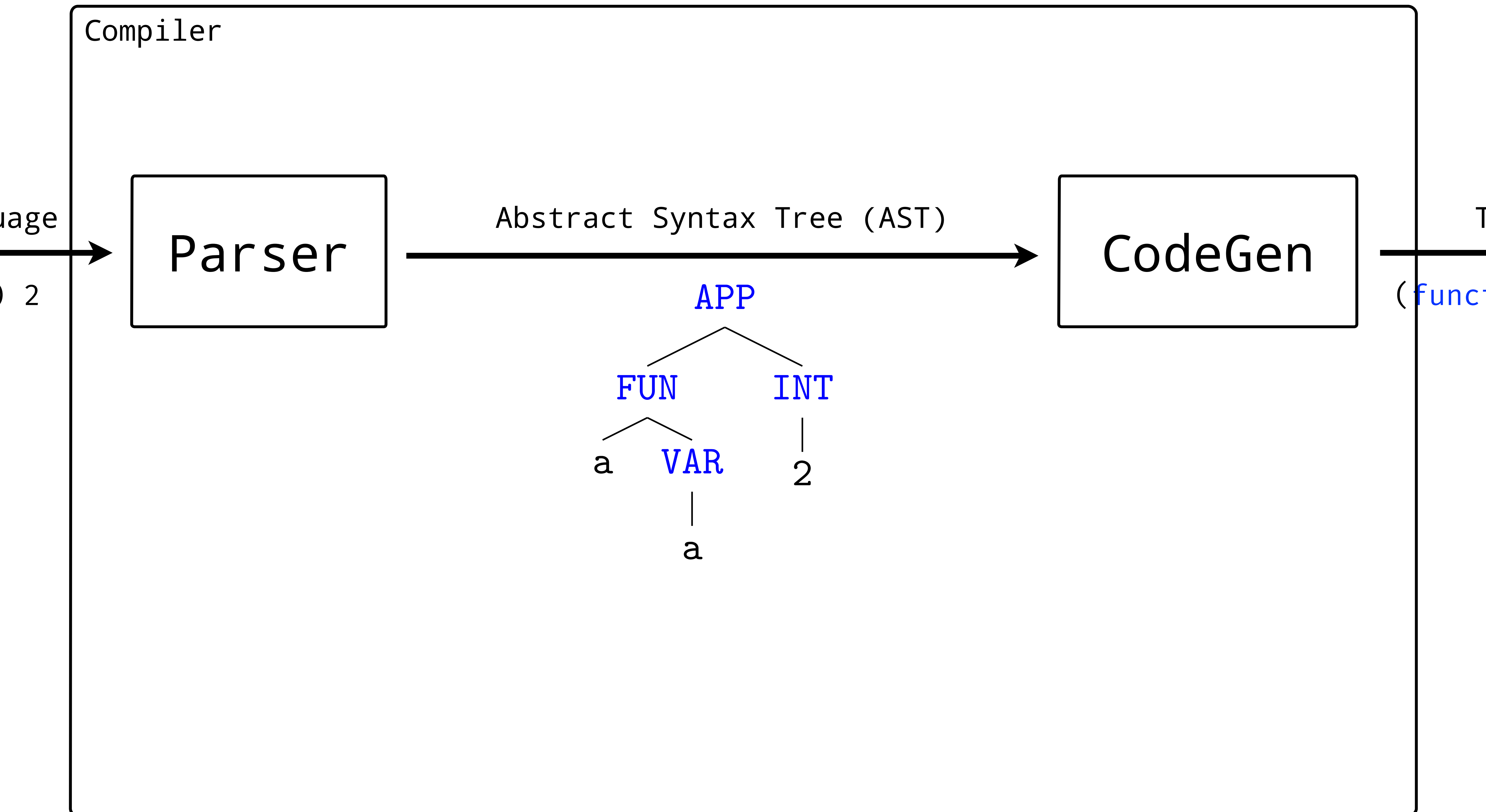


Parsing

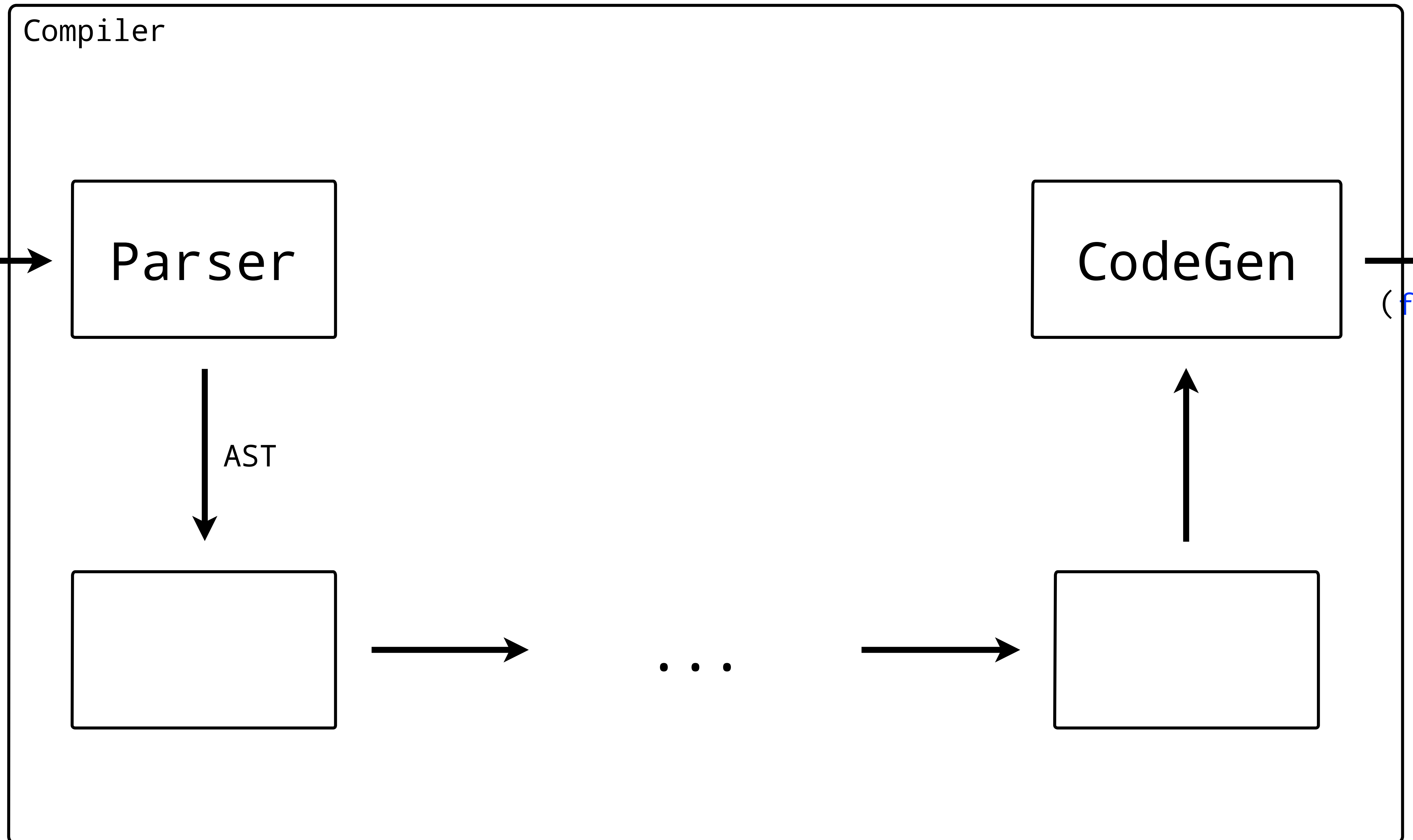
Abstract Syntax Tree



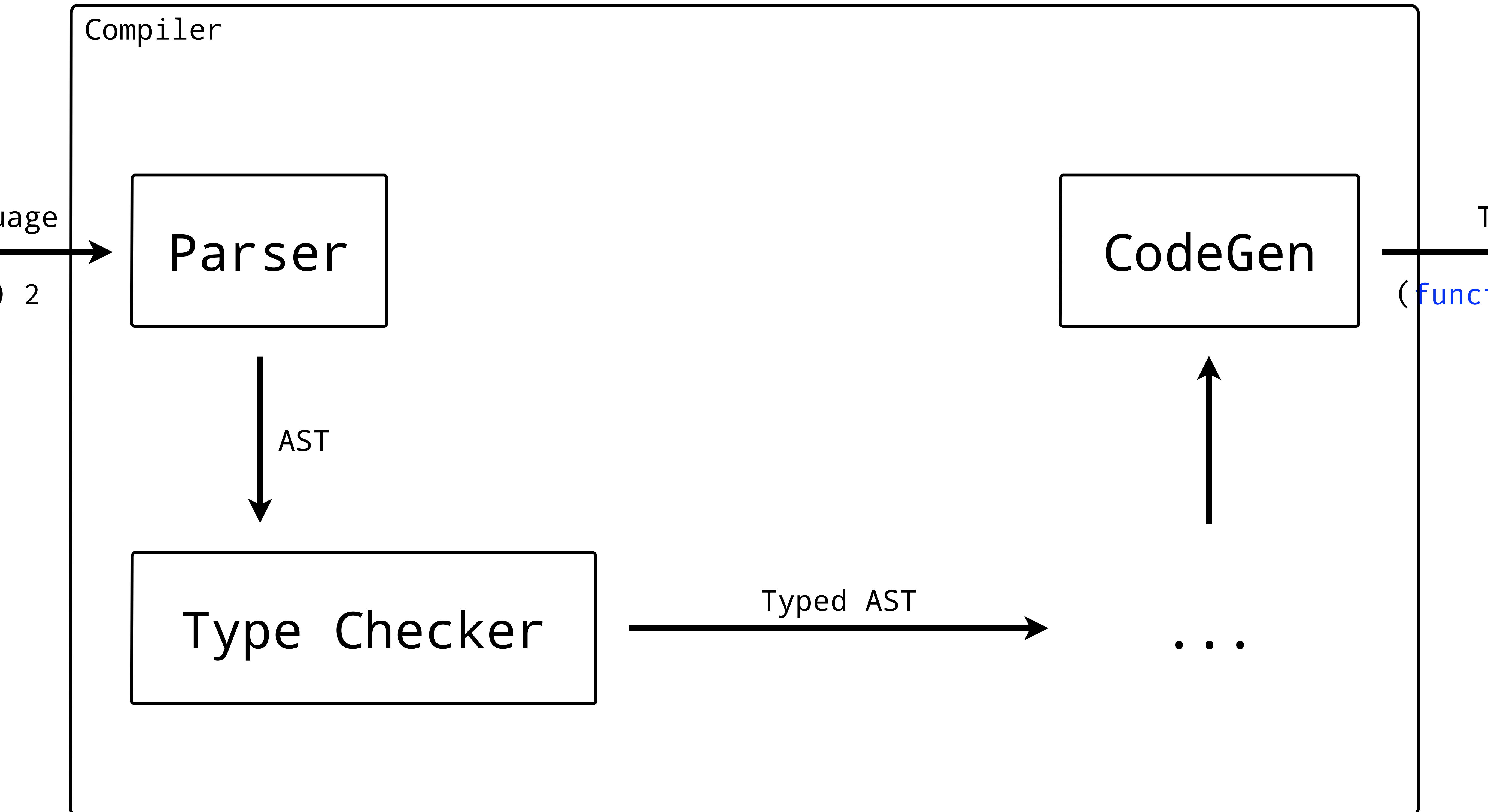
Code Generation



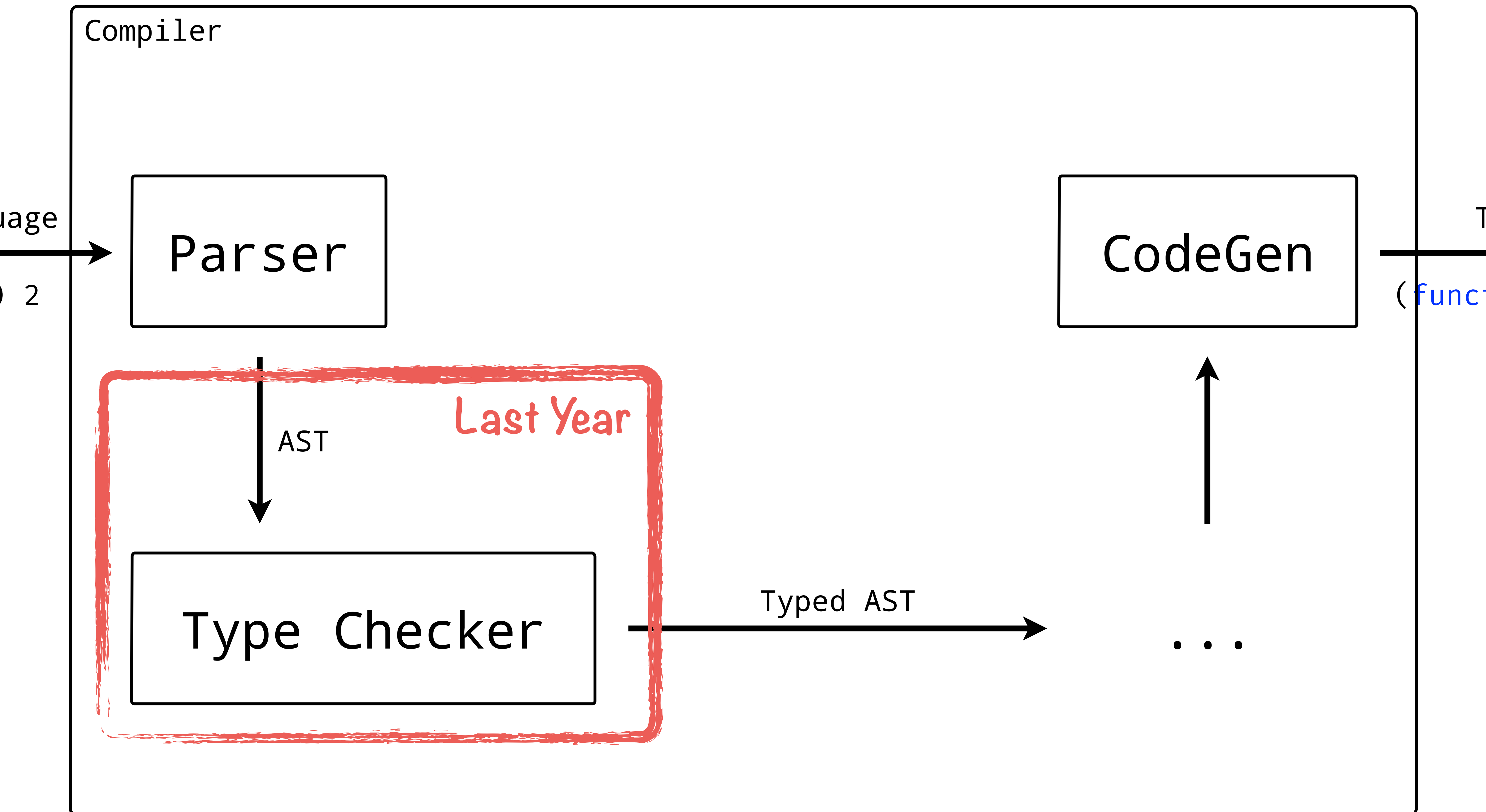
Many Intermediate Phases



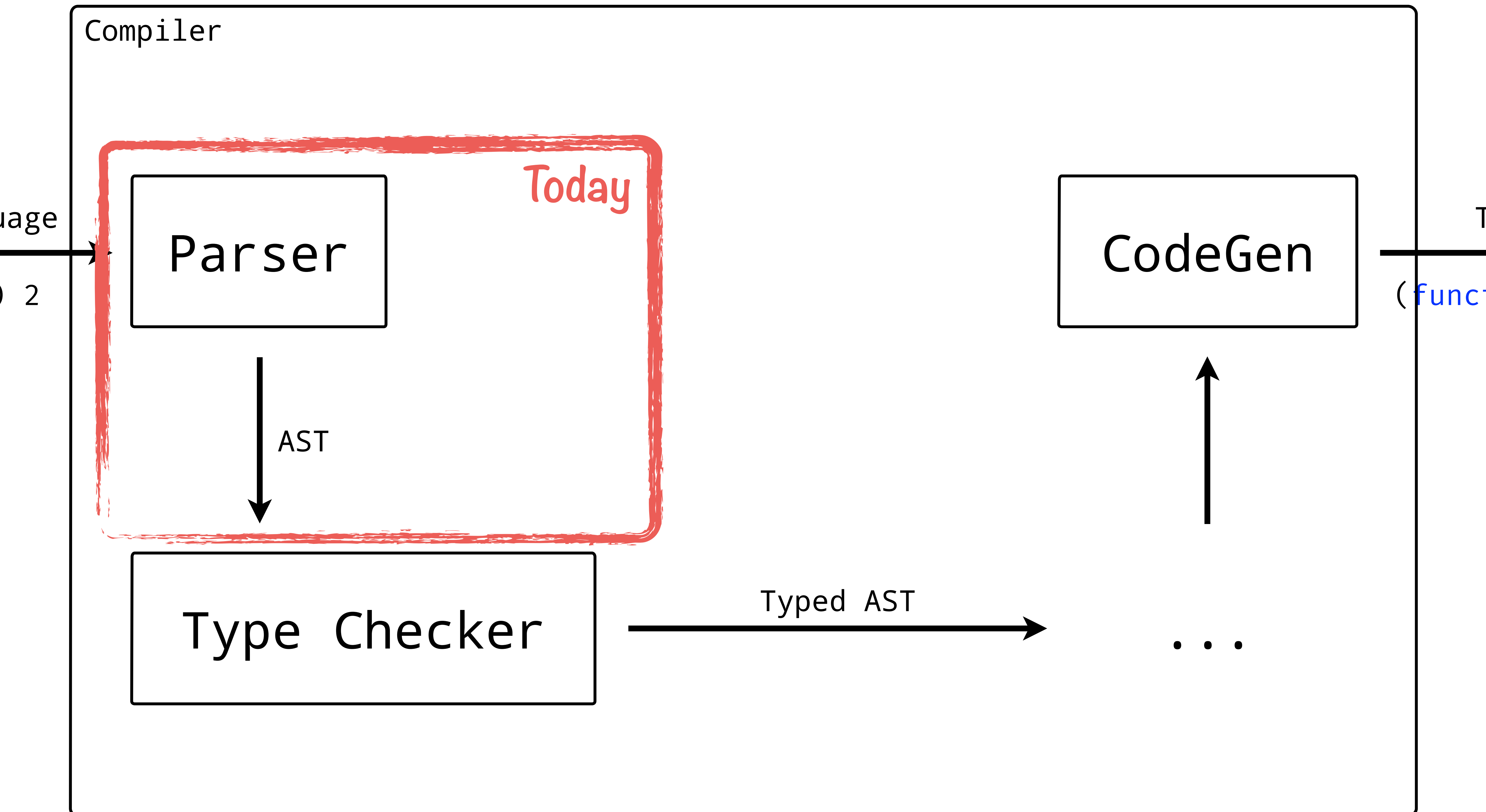
Type Checking



Last Year's Talk

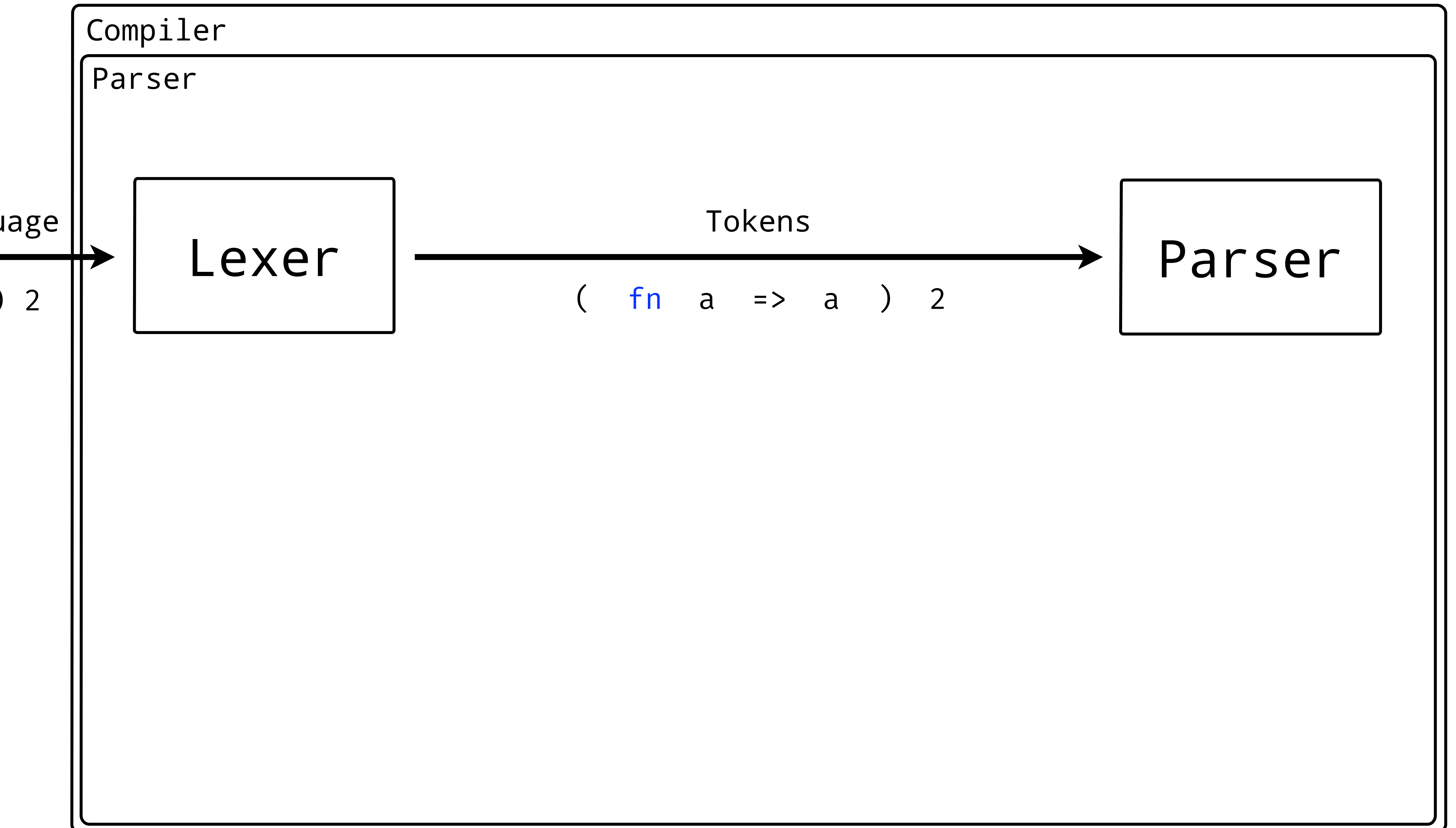


Today's Talk

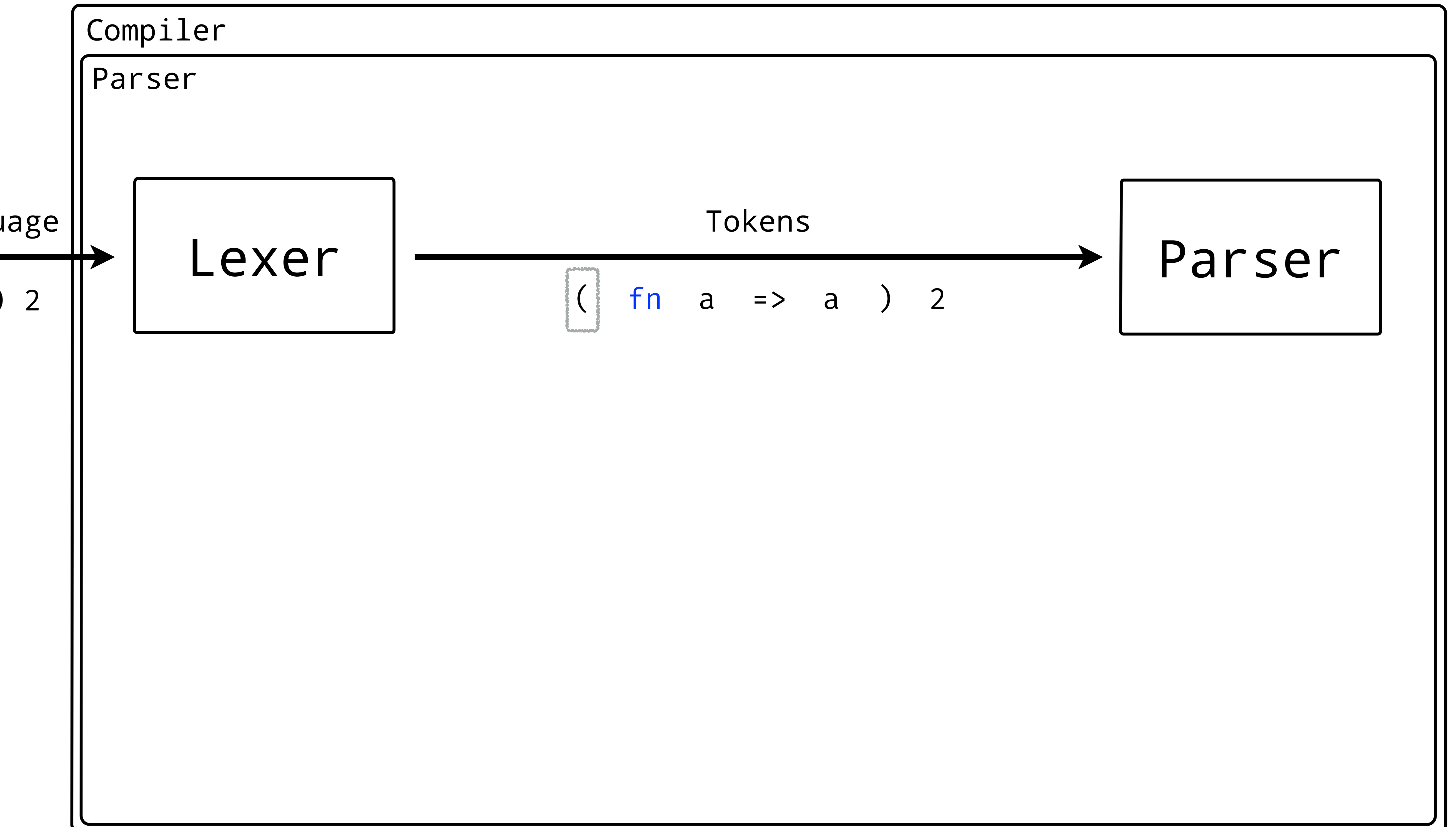


Parsing

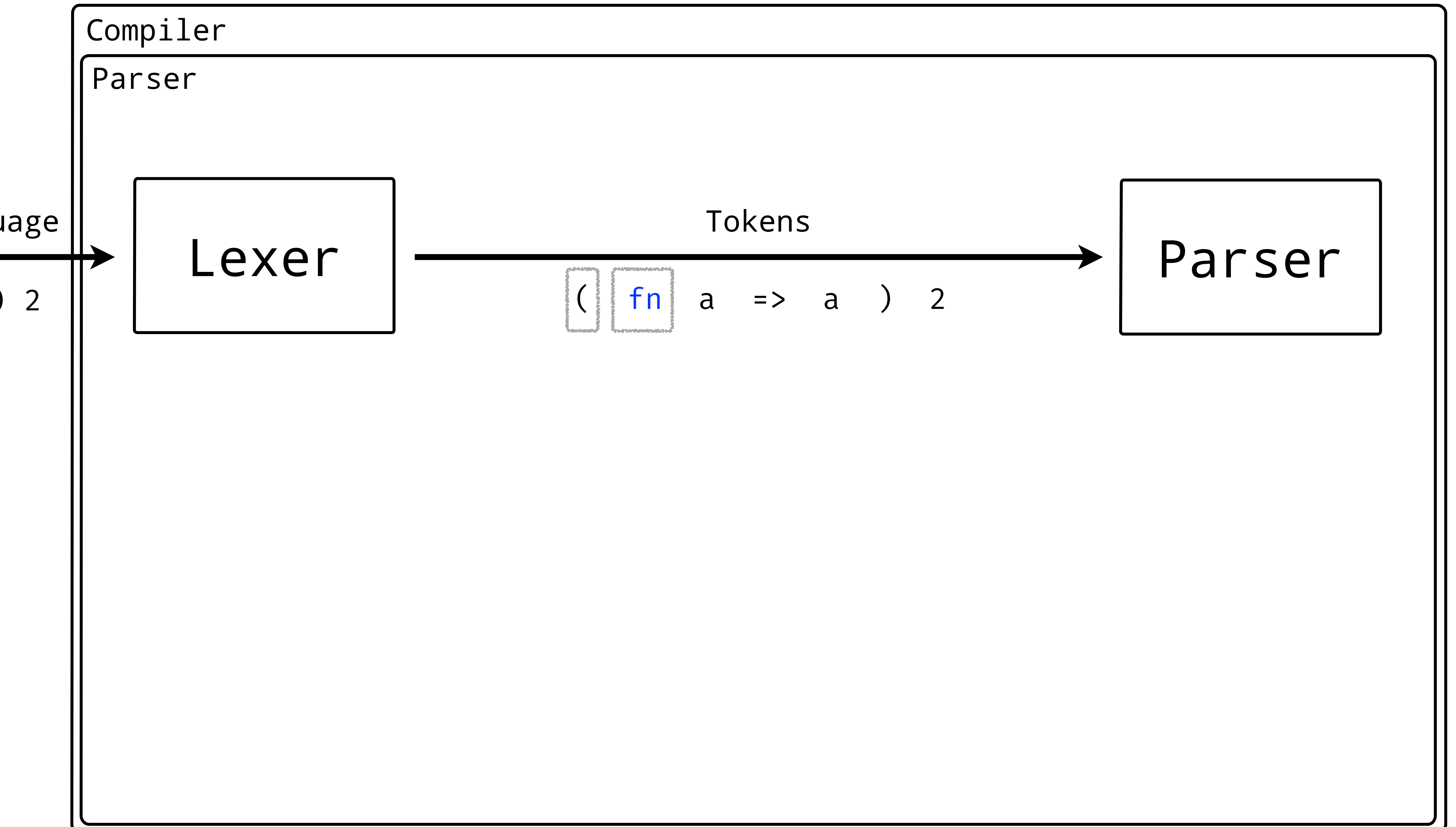
Lexing + Parsing



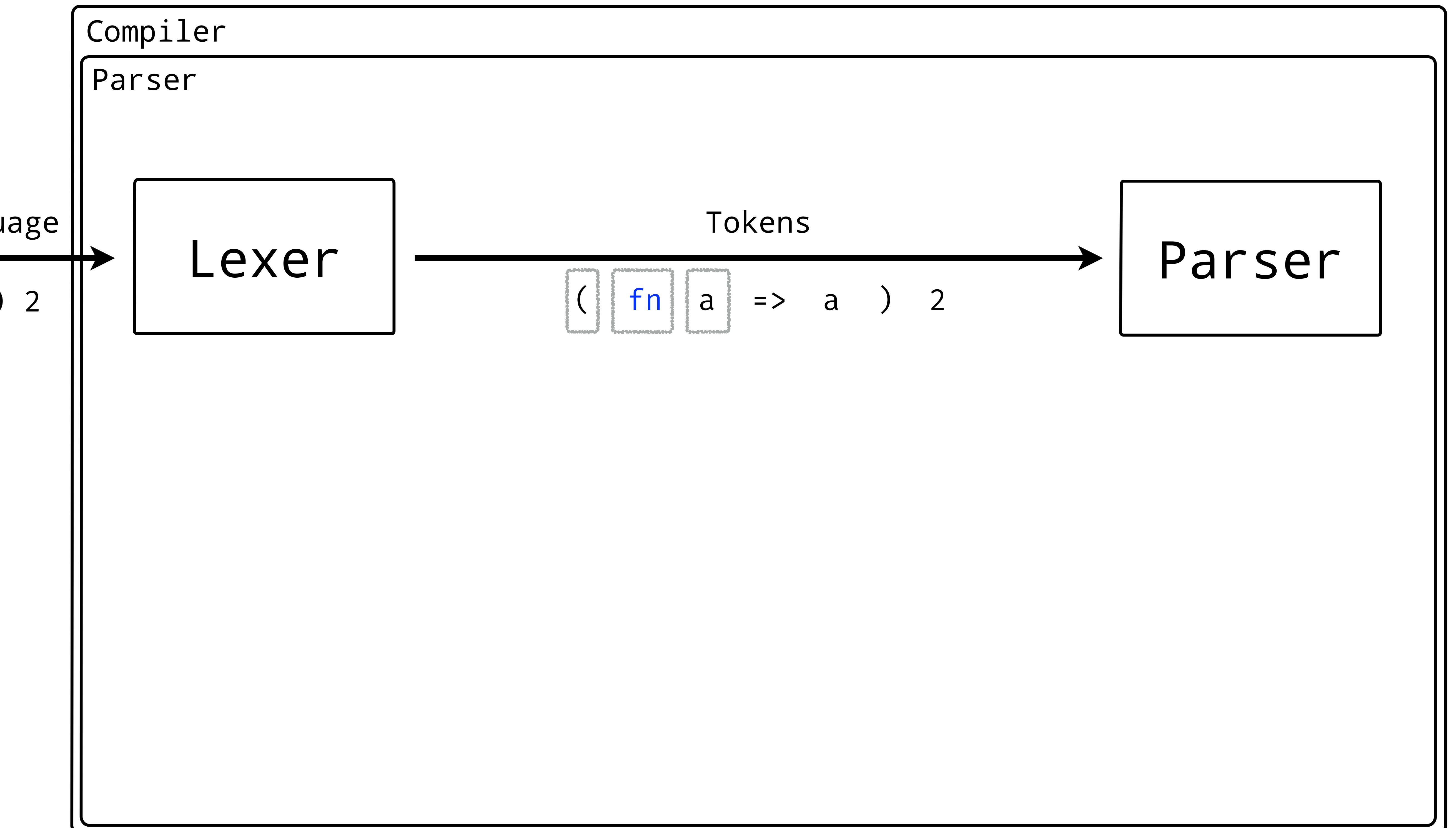
Lexing



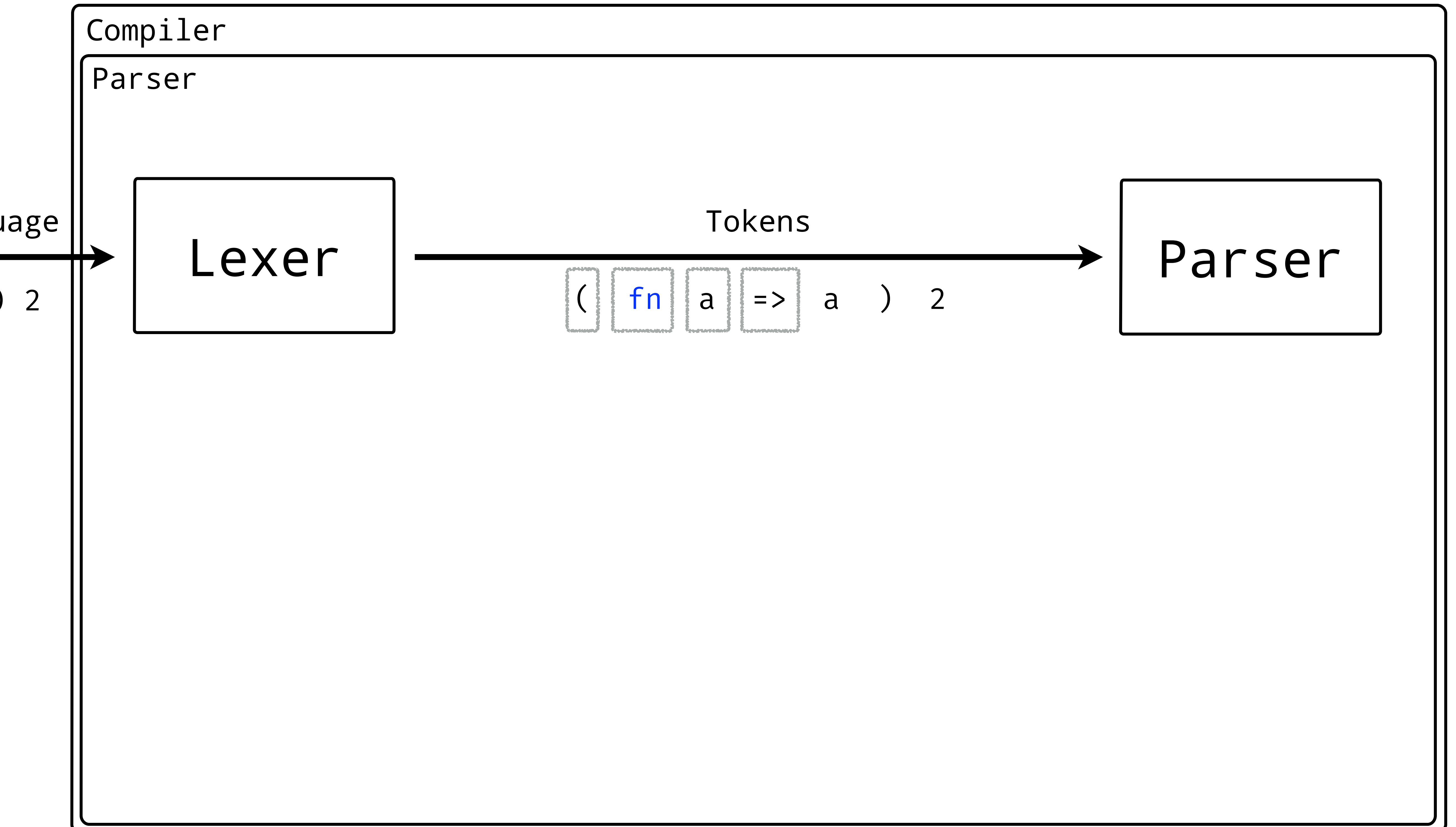
Lexing



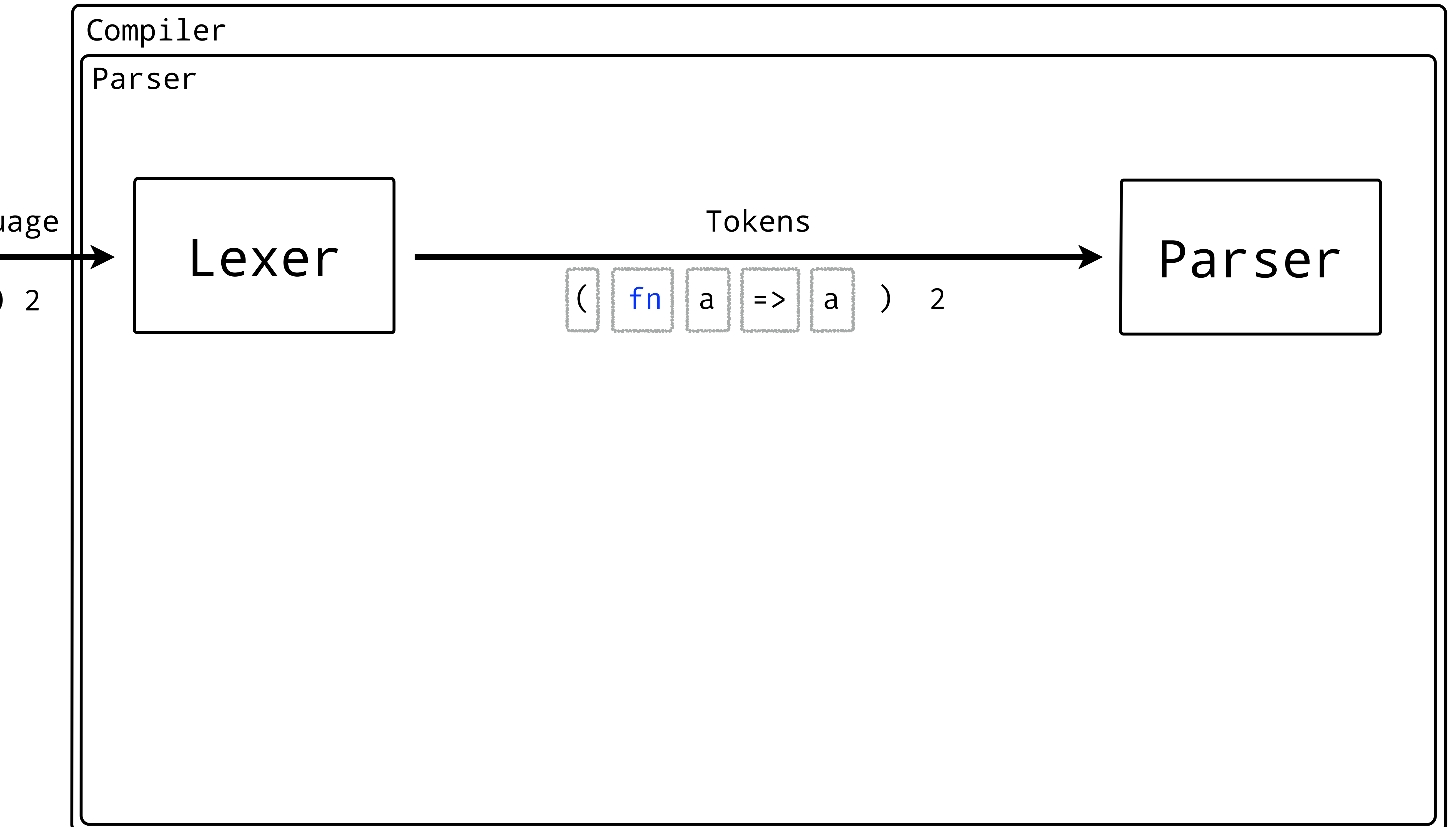
Lexing



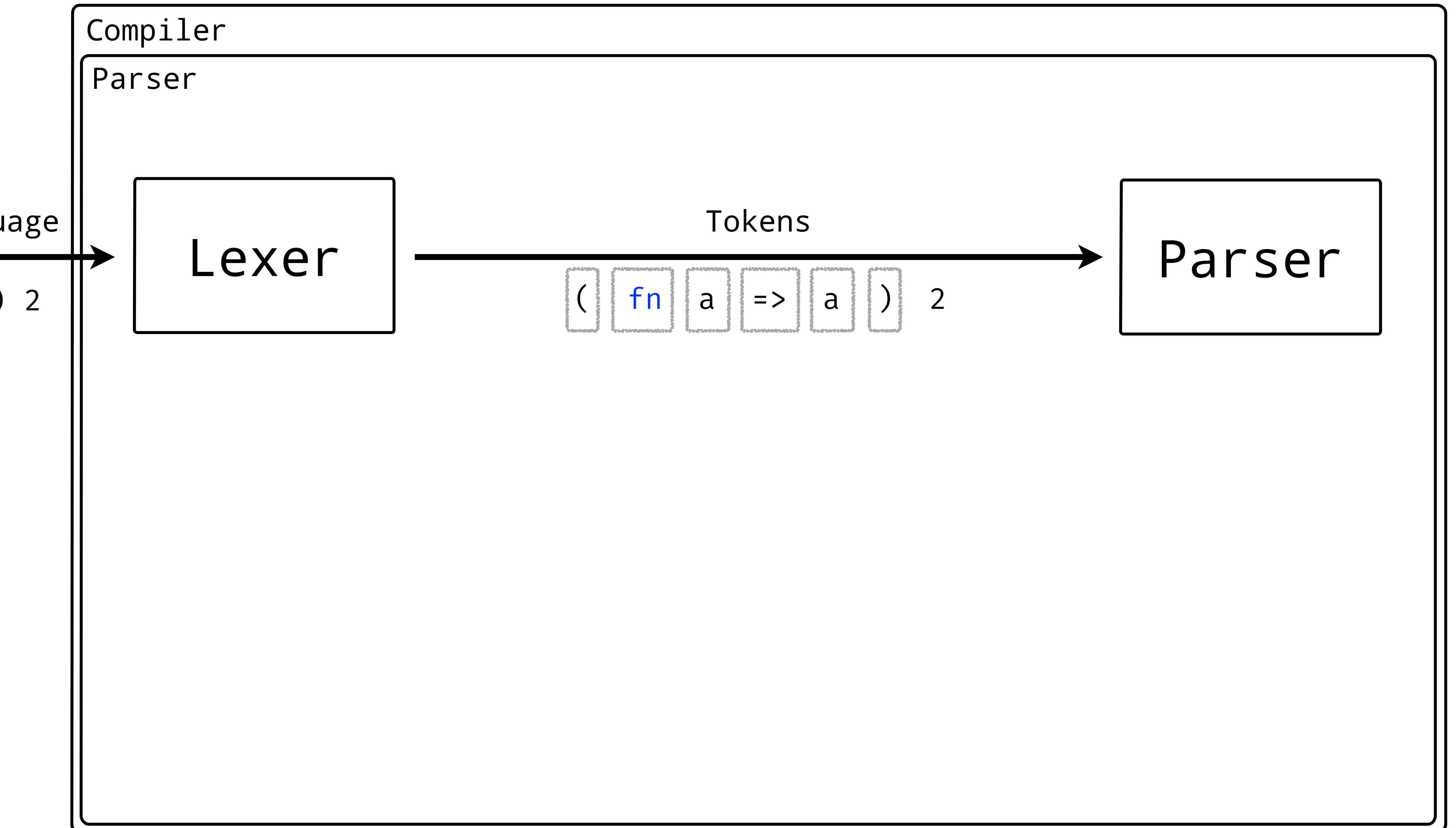
Lexing



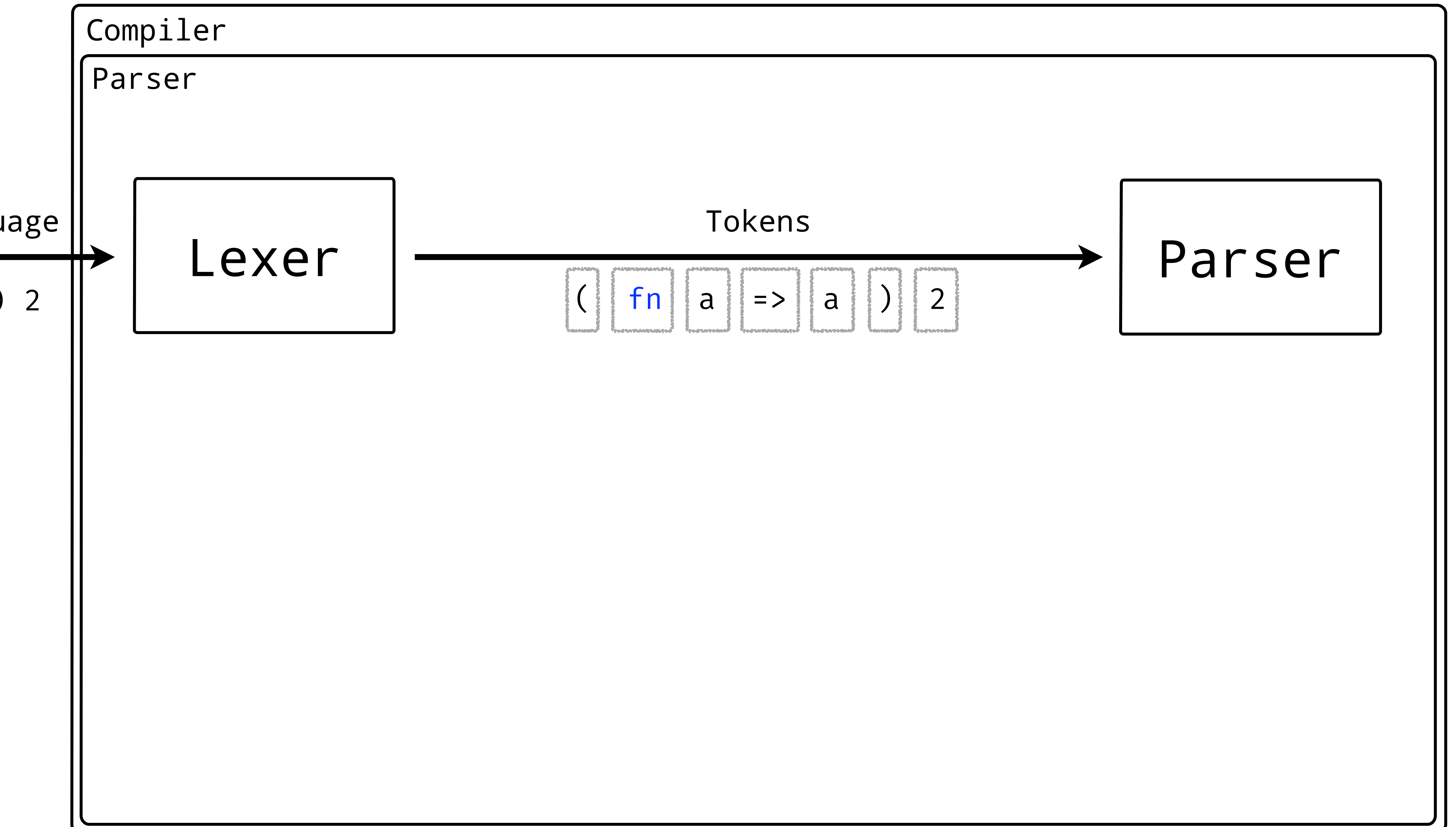
Lexing



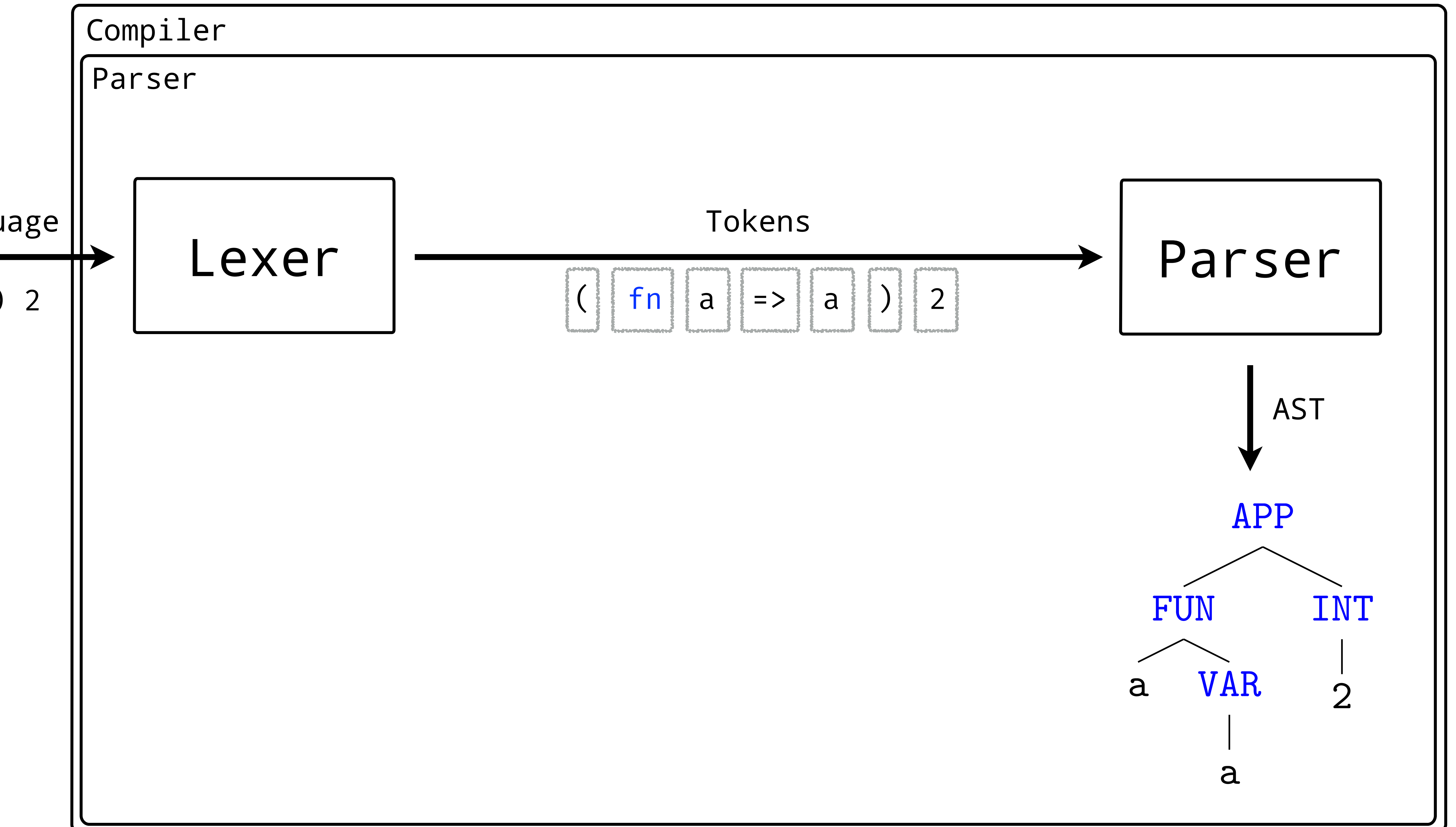
Lexing



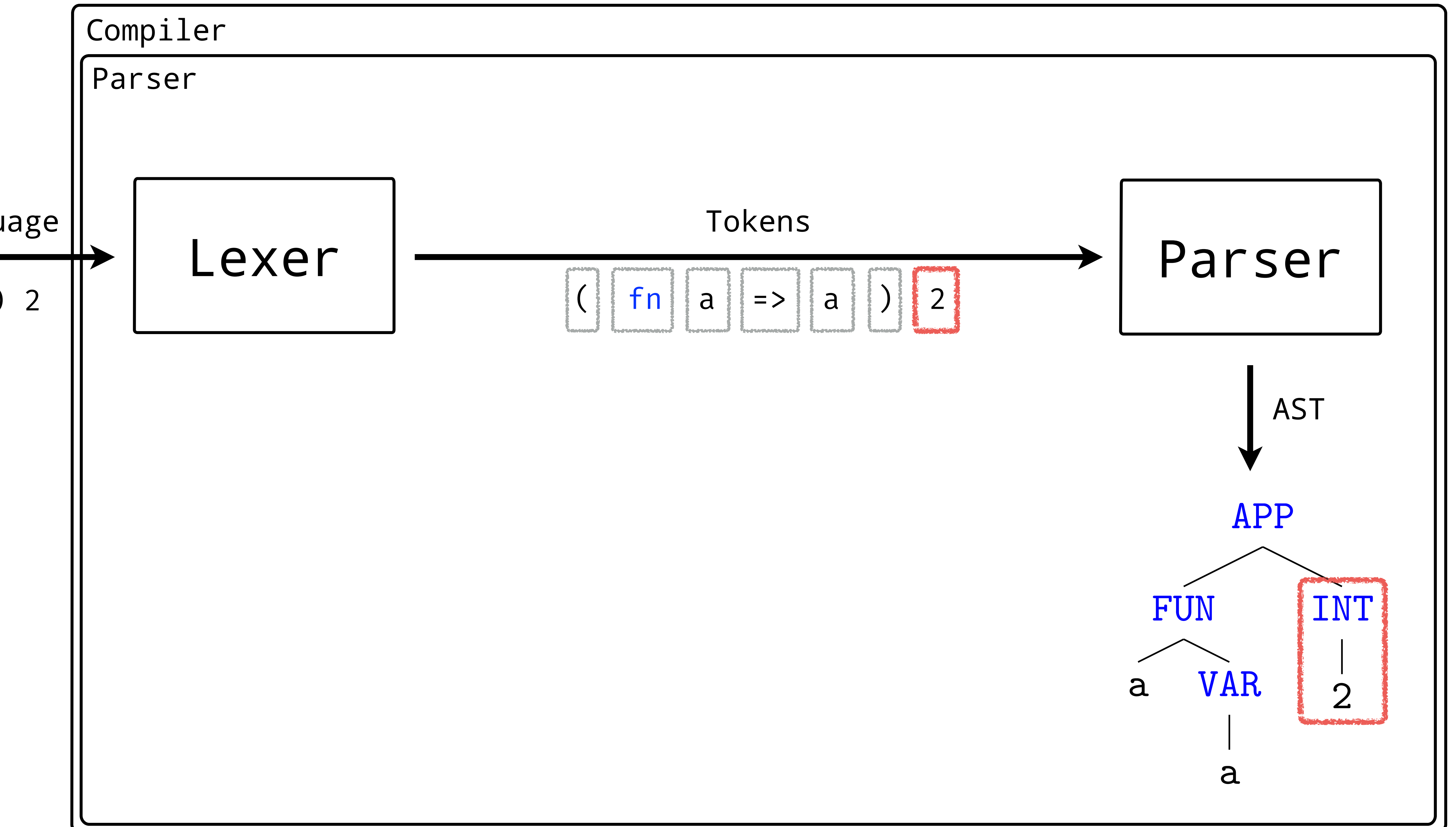
Lexing



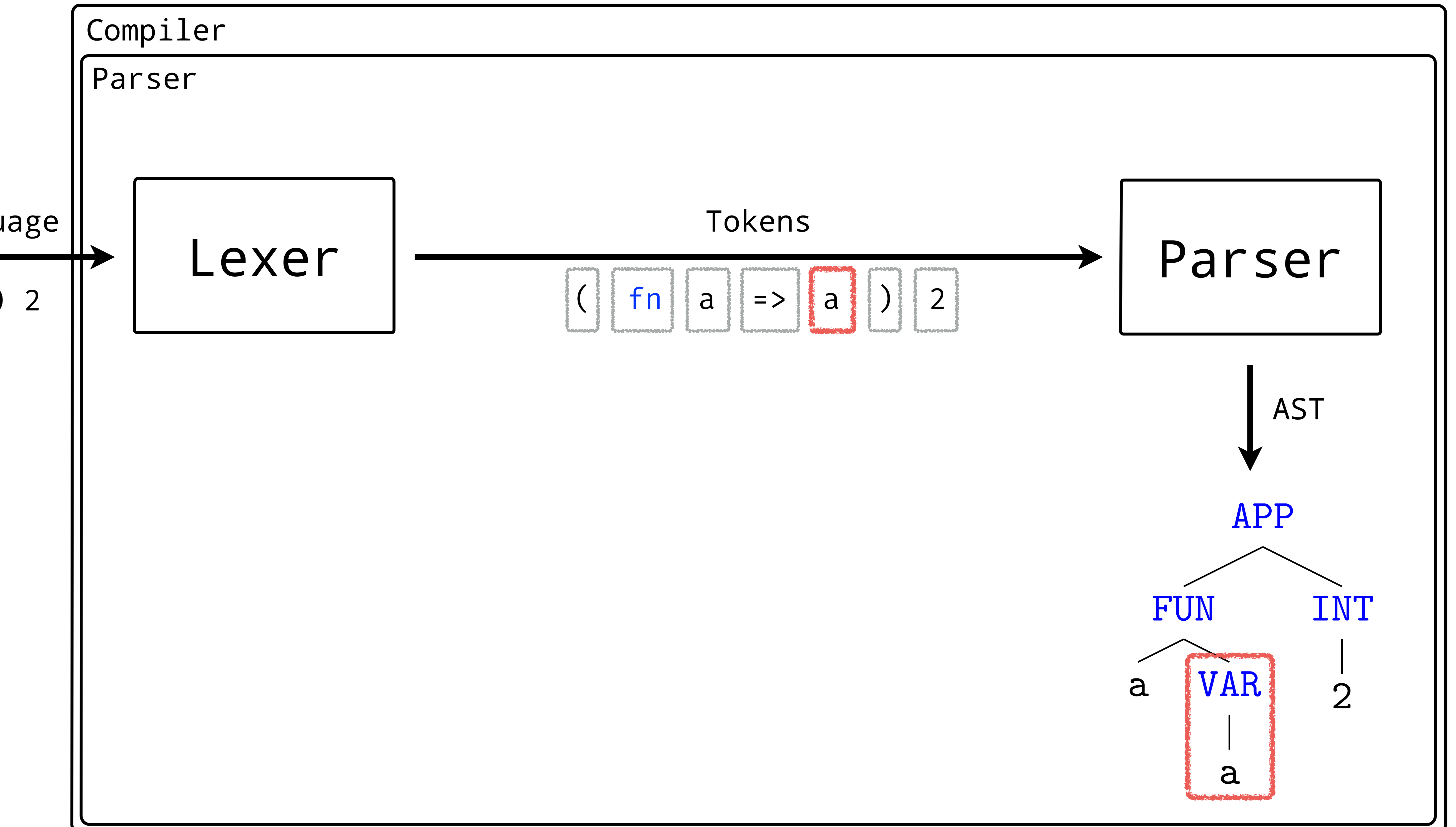
Lexing



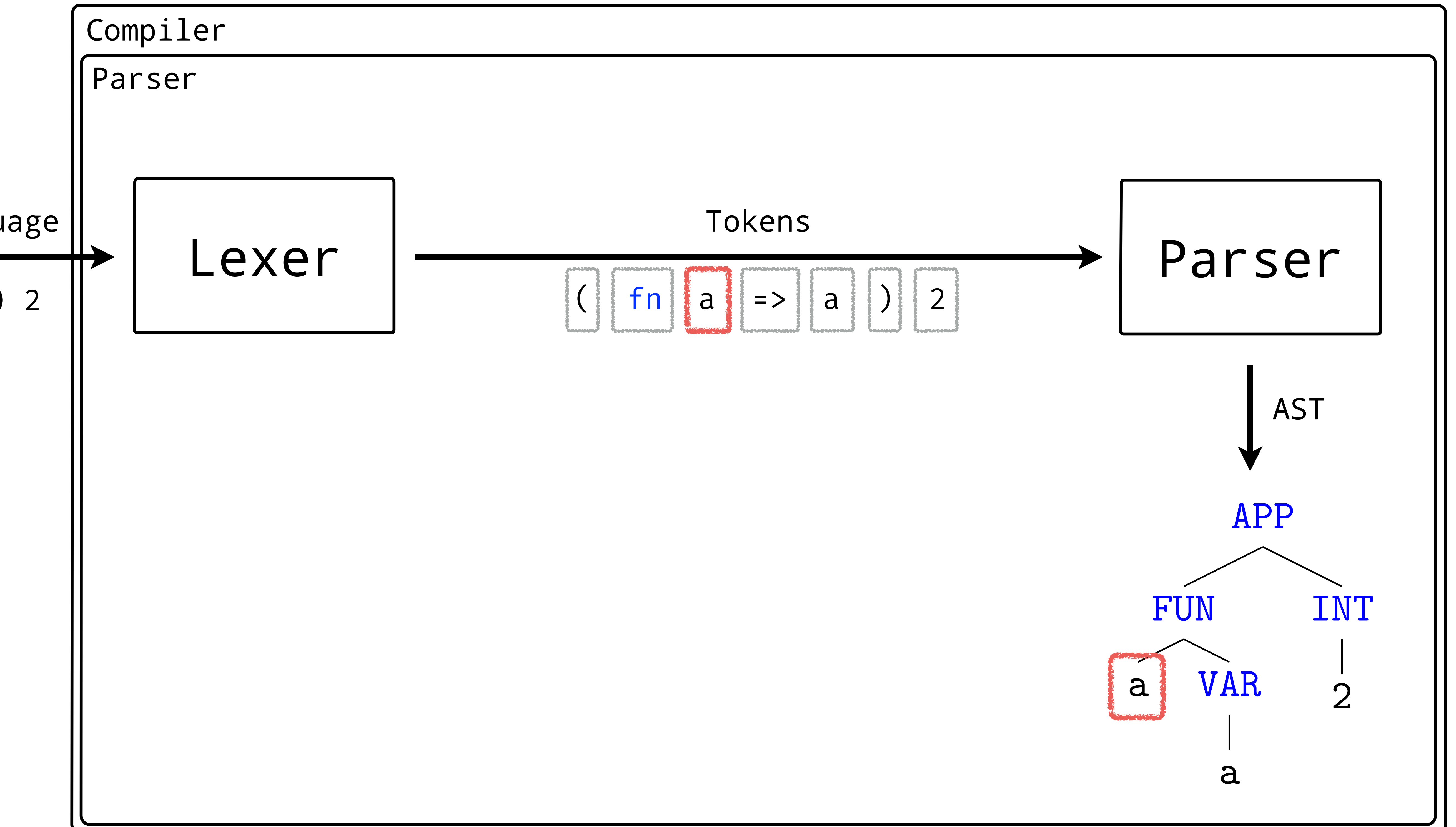
Parsing



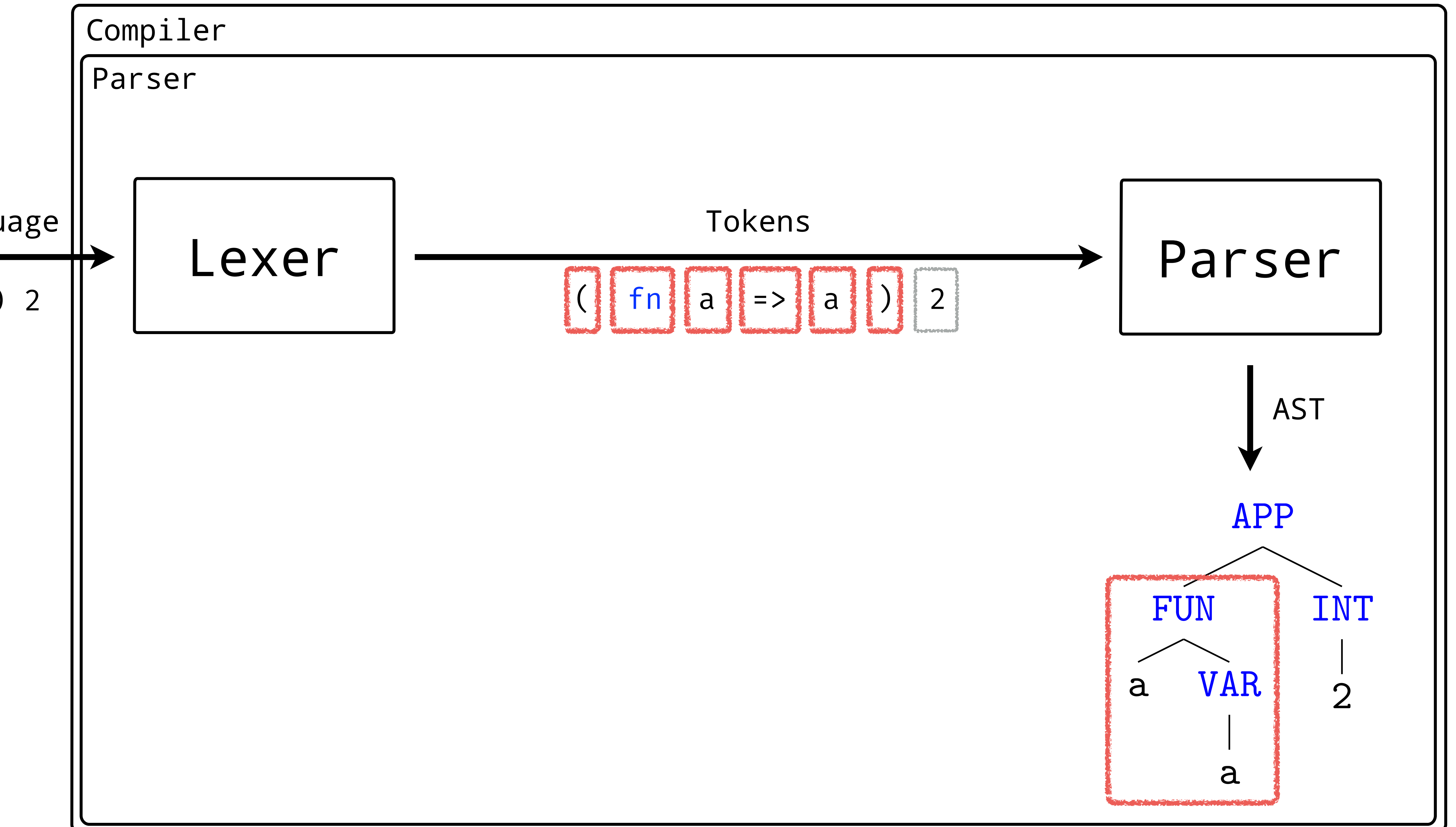
Parsing



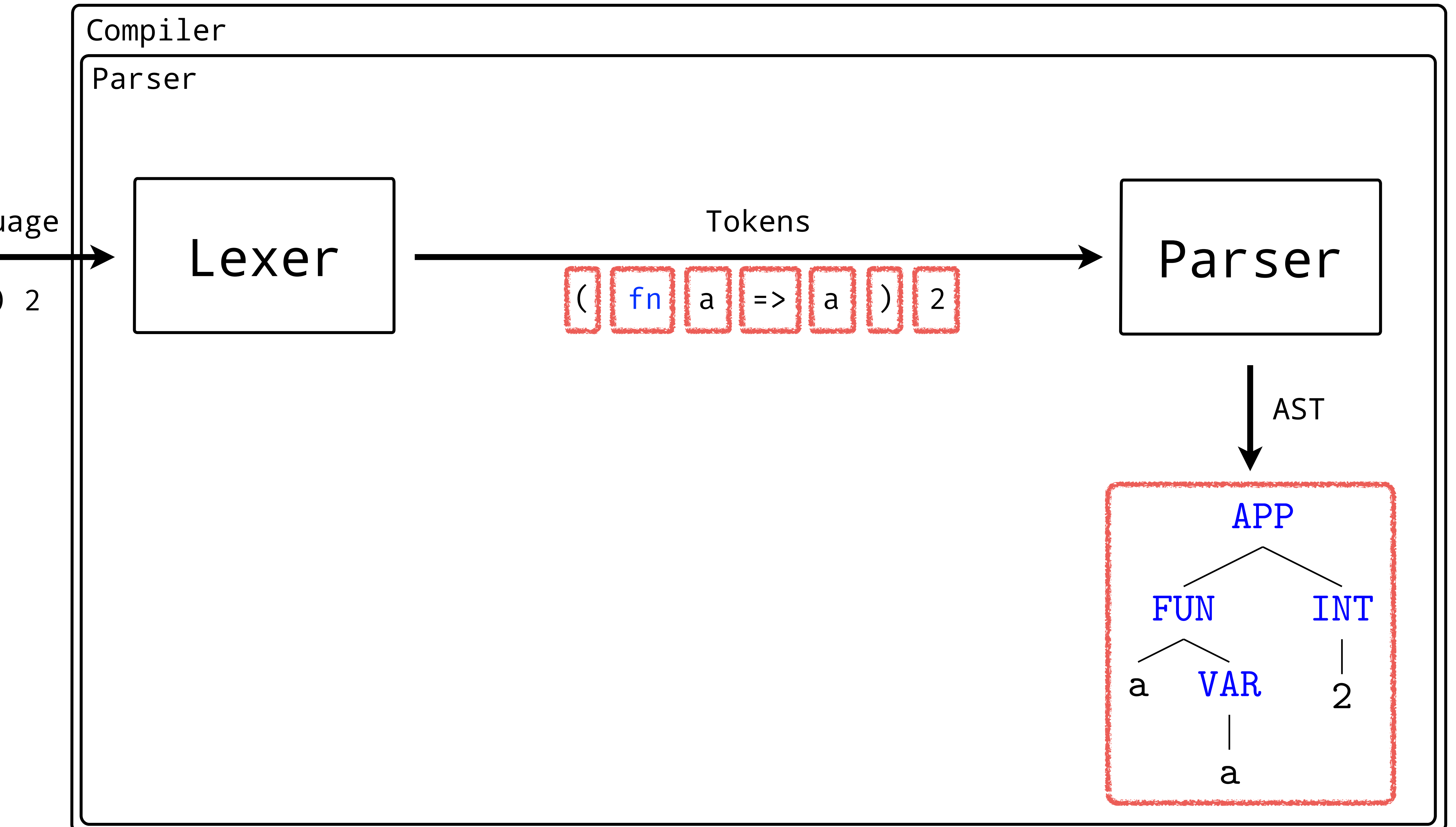
Parsing



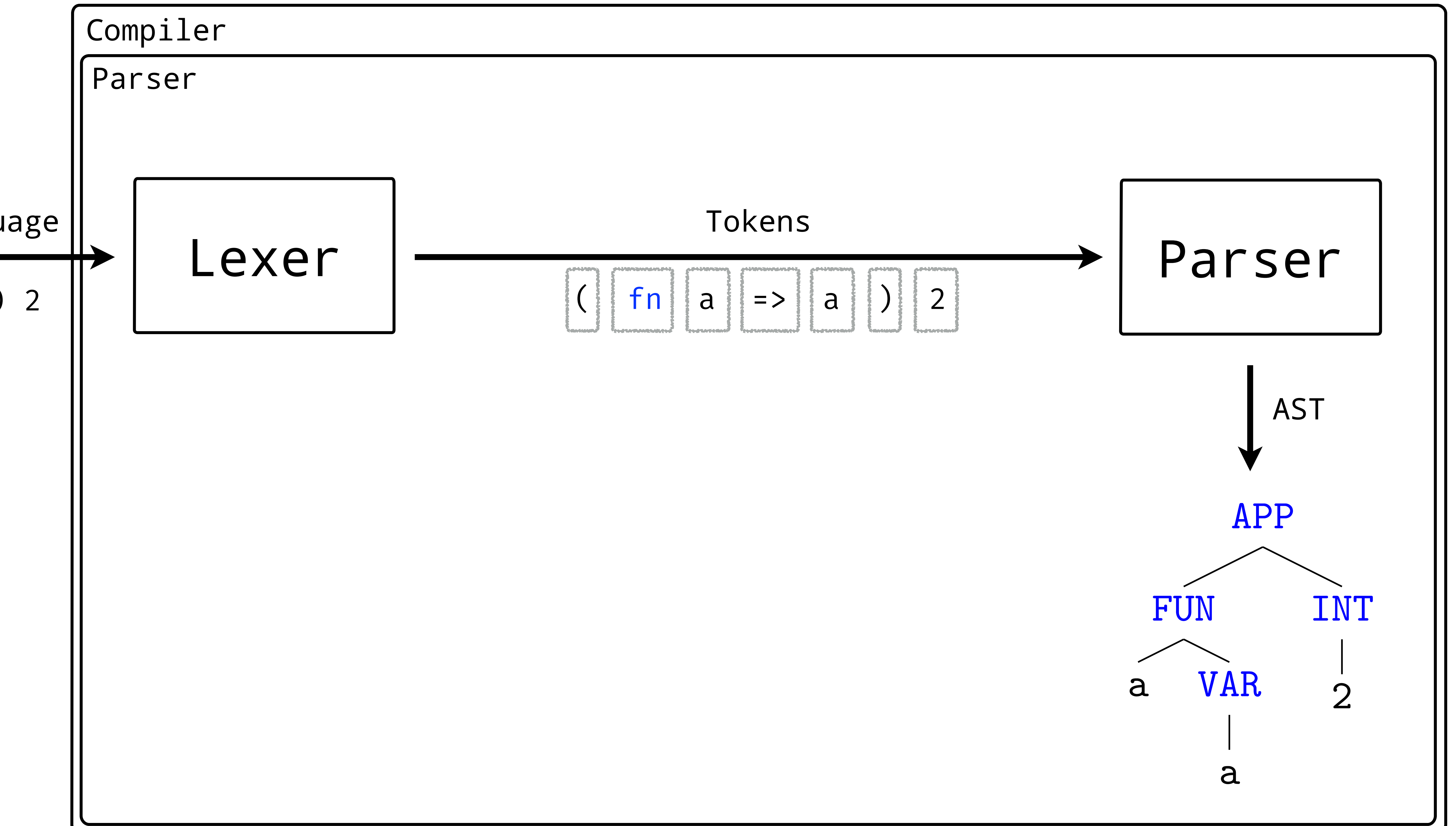
Parsing



Parsing

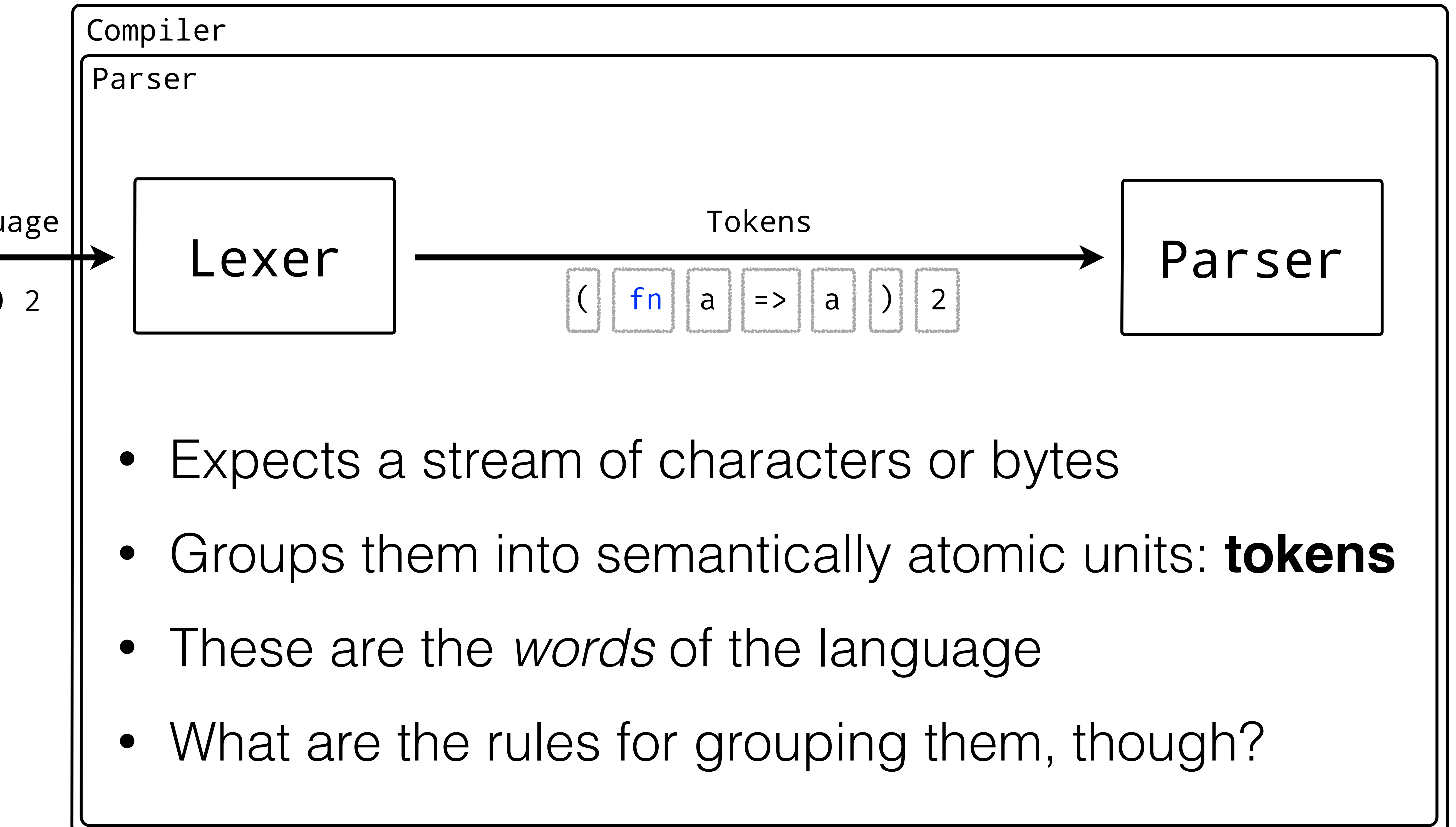


Lexing



Lexing

Lexing



- Expects a stream of characters or bytes
- Groups them into semantically atomic units: **tokens**
- These are the *words* of the language
- What are the rules for grouping them, though?

Lexing

- Grouping can be thought of as "split by space"

Lexing

- Grouping can be thought of as "split by space"
- Why not exactly that, though? Consider:

Lexing

- Grouping can be thought of as "split by space"
- Why not exactly that, though? Consider:

```
val sum = 1 + 2

val sum=1+2

val str = "spaces matter here"
```

Lexing

- We need rules for grouping characters into tokens

Lexing

- We need rules for grouping characters into tokens
- These rules form the **lexical grammar**

Lexing

- We need rules for grouping characters into tokens
- These rules form the **lexical grammar**
- Can be defined using regular expressions

Lexing

- We need rules for grouping characters into tokens
- These rules form the **lexical grammar**
- Can be defined using regular expressions
- Conducive to easy and efficient implementations

Lexing

- We need rules for grouping characters into tokens
- These rules form the **lexical grammar**
- Can be defined using regular expressions
- Conducive to easy and efficient implementations
  - Using a RegExp library

Lexing

- We need rules for grouping characters into tokens
- These rules form the **lexical grammar**
- Can be defined using regular expressions
- Conducive to easy and efficient implementations
  - Using a RegExp library
  - By hand isn't hard either, just a little cumbersome

Lexing

- We need rules for grouping characters into tokens
- These rules form the **lexical grammar**
- Can be defined using regular expressions
- Conducive to easy and efficient implementations
    - Using a RegExp library
    - By hand isn't hard either, just a little cumbersome
    - Lexer generators: Lex, Flex, Alex, ANTLR, etc.

Lexing

- We need rules for grouping characters into tokens
- These rules form the **lexical grammar**
- Can be defined using regular expressions
- Conducive to easy and efficient implementations
  - Using a RegExp library
  - By hand isn't hard either, just a little cumbersome
  - Lexer generators: Lex, Flex, Alex, ANTLR, etc.
- Lexing is what you need for syntax definition files

µML — Lexical Grammar

integers	
identifiers	
symbols	
keywords	

μML — Lexical Grammar

integers	`0|[1-9][0-9]*`
identifiers	
symbols	
keywords	

μML — Lexical Grammar

integers	`0|[1-9][0-9]*`
identifiers	`[a-zA-Z]+`
symbols	
keywords	

μML — Lexical Grammar

integers	`0|[1-9][0-9]*`
identifiers	`[a-zA-Z]+`
symbols	`(`, `)`, `+`, `-`, `=`, `=>`
keywords	

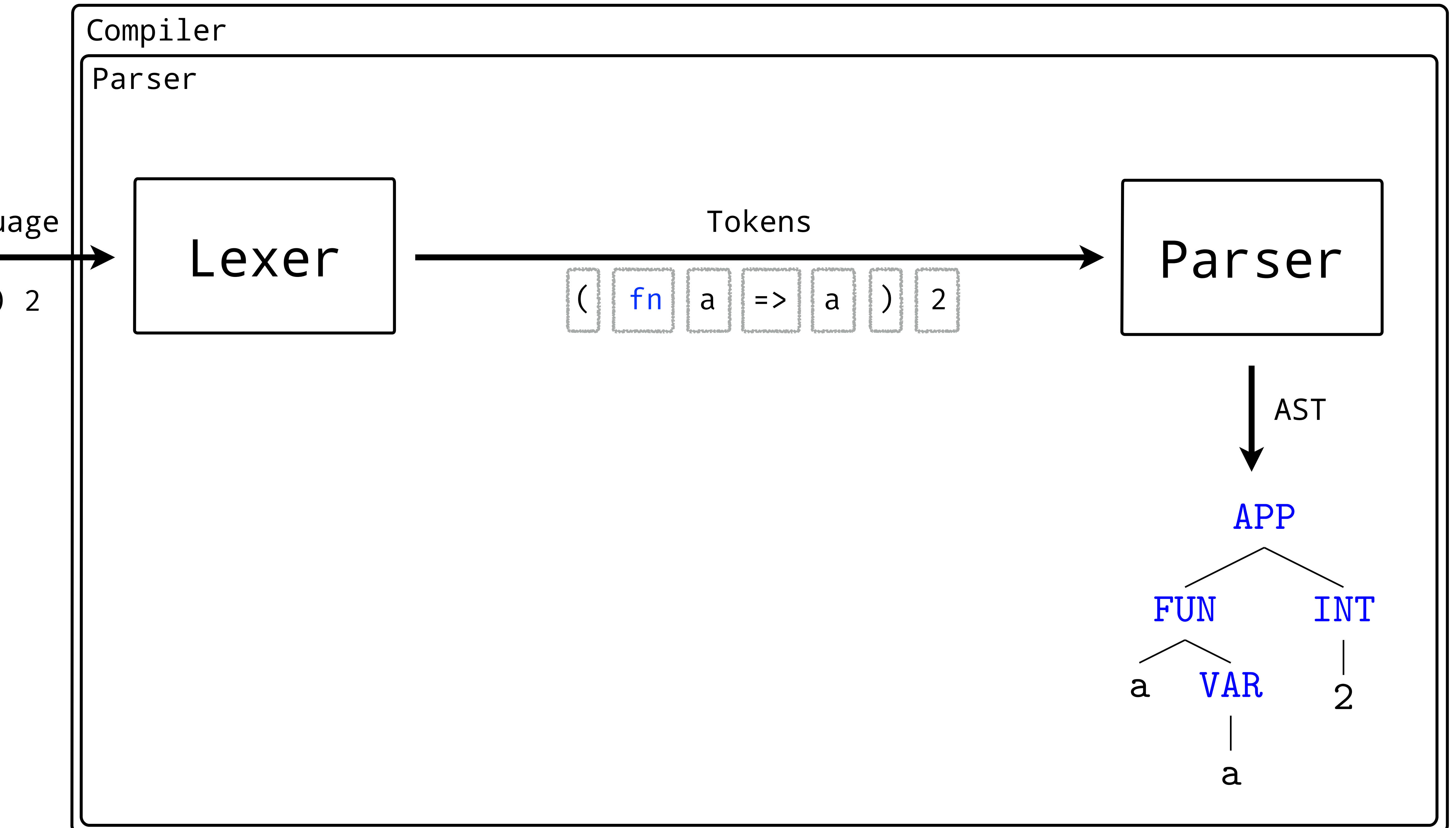
μML — Lexical Grammar

integers	`0|[1-9][0-9]*`
identifiers	`[a-zA-Z]+`
symbols	`(`, `)`, `+`, `-`, `=`, `=>`
keywords	`if`, `then`, `else`, `let`, `val`, `in`, `end`, `fn`, `true`, `false`

Code

Parsing

Parsing



Parsing

- The lexer recognizes valid words in the language

Parsing

- The lexer recognizes valid words in the language
- Not all combinations of valid words form valid phrases in a language

Parsing

- The lexer recognizes valid words in the language
- Not all combinations of valid words form valid phrases in a language
- Syntactically correct: `val a = 1`

Parsing

- The lexer recognizes valid words in the language
- Not all combinations of valid words form valid phrases in a language
- Syntactically correct: `val a = 1`
- Syntactically incorrect: `val val val`

Parsing

- The lexer recognizes valid words in the language
- Not all combinations of valid words form valid phrases in a language
- Syntactically correct: `val a = 1`
- Syntactically incorrect: `val val val`
- We must define the structure of phrases

Parsing

- The lexer recognizes valid words in the language
- Not all combinations of valid words form valid phrases in a language
- Syntactically correct: `val a = 1`
- Syntactically incorrect: `val val val`
- We must define the structure of phrases
- A syntactical grammar achieves that

Parsing

- Regular expressions are not powerful enough

Parsing

- Regular expressions are not powerful enough
- REs can't recognize nested structures

Parsing

- Regular expressions are not powerful enough
- REs can't recognize nested structures
- Because they use a finite amount of memory

Parsing

- Regular expressions are not powerful enough
- REs can't recognize nested structures
- Because they use a finite amount of memory
- Nesting needs a stack to remember the upper structures you're traversing

Parsing

- Regular expressions are not powerful enough
- REs can't recognize nested structures
- Because they use a finite amount of memory
- Nesting needs a stack to remember the upper structures you're traversing
- Syntactical grammars express nesting using recursion

You can't parse [X]HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. so many times but it is not getting to me. Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing HTML. You will never make me crack. HTML is a language of sufficient complexity that it cannot be parsed by regular expressions. Even Jon Skeet cannot parse HTML using regular expressions. Every time you attempt to parse HTML with regular expressions, the unholy child weeps the blood of virgins, and Russian hackers pwn your webapp. Parsing HTML with regex summons tainted souls into the realm of the living. HTML and regex go together like love, marriage, and ritual infanticide. The <center> cannot hold it is too late. The force of regex and HTML together in the same conceptual space will destroy your mind like so much watery putty. If you parse HTML with regex you are giving in to Them and their blasphemous ways which doom us all to inhuman toil for the One whose Name cannot be expressed in the Basic Multilingual Plane, he comes. HTML-plus-regexp will liquify the nerves of the sentient whilst you observe, your psyche withering in the onslaught of horror. Regex-based HTML parsers are the cancer that is killing StackOverflow *it is too late it is too late we cannot be saved* the trangession of a child ensures regex will consume all living tissue (except for HTML which it cannot, as previously prophesied) *dear lord help us how can anyone survive this scourge* using regex to parse HTML has doomed humanity to an eternity of dread torture and security holes *using regex* as a tool to process HTML establishes a brea*ch between this world* and the dread realm of corrupt entities (like SGML entities, but *more corrupt) a mere glimp*se of the world of reg**ex parsers for HTML will ins**tantly transport a p*rogrammer's consciousness i*nto a wor*l*d of ceaseless screaming, he comes~~, the pestilent s~~lithy regex-infection wil**l devour your HT**ML parser, application and existence for all time like Visual Basic only worse *he comes he comes do not fi*ght h**e comes, hi**s unholy radiancé de*stroying all enlightenment, HTML tags* ***leaking from your eyes like liq****uid p*ain, the song of re*gular expre~~ssion parsing~~ will ext*inguish the voices of mor***tal man from the sp***here I can see it can you see it it is beautiful t*he f `inal snuf` fing o*f the lie**s of Man ALL IS LOST ALL I***S L*O*ST the p*ony he c*o*mes he ~~comes he~~ co**mes t*he* ich**or permeat*es al*l MY FACE MY FACE ᵒh god n***o NO NOOOO N**Θ stop t*he an*gles `are` **not real ZALGO IS** TONY THE **PONY HE COMES**

Have you tried using an XML parser instead?

You can't parse [X]HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. so many times but it is not getting to me. Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing HTML. You will never make me crack. HTML is a language of sufficient complexity that it cannot be parsed by regular expressions. Even Jon Skeet cannot parse HTML using regular expressions. Every time you attempt to parse HTML with regular expressions, th... weeps the blood of virgins, and Russian hackers pwn your... summons tainted souls into the real... and ritu...

It's not weird-looking Unicode characters that make regexes unsuitable for parsing.

...more corrupt) a ...instantly transport a *programmer's* ...ceaseless screaming, he comes, ~~the pestilent slithy regex-infection~~ ...your HTML parser, application and existence for all time like Visual Basic only worse *he comes he comes do not fi*ght h**e comes, hi**s unholy radiancé dest*roying all enlightenment, HTML tags* ***leaking from your eyes like liq****uid* pain, the song of regular expre~~ssion parsing~~ will exti*nguish the voices of mor****tal man from the sp****here I can see it can you see it it is beautiful t*he f `inal snuf` fing o*f the lie****s of Man ALL IS LOST ALL I****S L*OST th*e pony he c*omes he c~~omes he~~ co**mes t*he ich***or permeates al*l MY FAC*E MY FACE ᵒh god n**o NO NOOOO N**Θ stop t*he an* *gl*es `a r e` n**ot rea**l **ZALGO IS** TONY THE **PONY HE COMES**

Have you tried using an XML parser instead?

Syntactical Grammar

µML — Syntactical Grammar

```
expr =
```

μML — Syntactical Grammar

```
expr = int
```

Here, blue symbols represent tokens coming from the lexer, not keywords.

μML — Syntactical Grammar

```
expr = int
     | var
```

Here, blue symbols represent tokens coming from the lexer, not keywords.

μML — Syntactical Grammar

```
expr = int
     | var
     | bool
```

Here, blue symbols represent tokens coming from the lexer, not keywords.

µML — Syntactical Grammar

```
expr = int
     | var
     | bool
```

```
bool = true | false
```

Here, blue symbols represent tokens coming from the lexer, not keywords.

µML — Syntactical Grammar

```
expr = int
     | var
     | bool
     | ( expr )

bool = true | false
```

Here, blue symbols represent tokens coming from the lexer, not keywords.

µML — Syntactical Grammar

```
expr = int
     | var
     | bool
     | ( expr )
     | fn var => expr

bool = true | false
```

Here, blue symbols represent tokens coming from the lexer, not keywords.

µML — Syntactical Grammar

```
expr = int
     | var
     | bool
     | ( expr )
     | fn var => expr
     | if expr then expr else expr

bool = true | false
```

Here, blue symbols represent tokens coming from the lexer, not keywords.

µML — Syntactical Grammar

```
expr = int
     | var
     | bool
     | ( expr )
     | fn var => expr
     | if expr then expr else expr
     | let val var = expr in expr end

bool = true | false
```

Here, blue symbols represent tokens coming from the lexer, not keywords.

µML — Syntactical Grammar

```
expr = int
     | var
     | bool
     | ( expr )
     | fn var => expr
     | if expr then expr else expr
     | let val var = expr in expr end
     | expr oper expr
```

```
bool = true | false
```

Here, blue symbols represent tokens coming from the lexer, not keywords.

µML — Syntactical Grammar

```
expr = int
     | var
     | bool
     | ( expr )
     | fn var => expr
     | if expr then expr else expr
     | let val var = expr in expr end
     | expr oper expr
```

```
bool = true | false
oper = + | -
```

Here, blue symbols represent tokens coming from the lexer, not keywords.

μML — Syntactical Grammar

```
expr = int
     | var
     | bool
     | ( expr )
     | fn var => expr
     | if expr then expr else expr
     | let val var = expr in expr end
     | expr oper expr
     | expr expr

bool = true | false
oper = + | -
```

Here, blue symbols represent tokens coming from the lexer, not keywords.

Introducing Precedence

- Function application has higher precedence over infix expressions in ML

Introducing Precedence

- Function application has higher precedence over infix expressions in ML
- double 1 + 2 **=** (double 1) + 2

Introducing Precedence

- Function application has higher precedence over infix expressions in ML
- double 1 + 2 **=** (double 1) + 2
- double 1 + 2 **≠** double (1 + 2)

Introducing Precedence

- Function application has higher precedence over infix expressions in ML
- double 1 + 2 **=** (double 1) + 2
- double 1 + 2 **≠** double (1 + 2)
- A rule's alternatives don't encode precedence

Introducing Precedence

- Function application has higher precedence over infix expressions in ML
- double 1 + 2 **=** (double 1) + 2
- double 1 + 2 **≠** double (1 + 2)
- A rule's alternatives don't encode precedence
- Grammars convey this by chaining rules in order of precedence

Introducing Precedence

- Function application has higher precedence over infix expressions in ML
- double 1 + 2 **=** (double 1) + 2
- double 1 + 2 **≠** double (1 + 2)
- A rule's alternatives don't encode precedence
- Grammars convey this by chaining rules in order of precedence
- Doesn't scale with many infix operators

Introducing Precedence

- Function application has higher precedence over infix expressions in ML
- double 1 + 2 **=** (double 1) + 2
- double 1 + 2 **≠** double (1 + 2)
- A rule's alternatives don't encode precedence
- Grammars convey this by chaining rules in order of precedence
- Doesn't scale with many infix operators
- Use a special parser for that, e.g., the Shunting Yard algorithm

Introducing Precedence

```
expr = int
     | var
     | bool
     | ( expr )
     | fn var => expr
     | if expr then expr else expr
     | let val var = expr in expr end
     | expr oper expr
     | expr expr
```

```
bool = true | false
oper = + | -
```

Introducing Precedence

```
expr =
```

```
bool = true | false
oper = + | -
```

```
atomic = int
       | var
       | bool
       | ( expr )
       | let val var = expr in expr end
```

Introducing Precedence

```
expr =

bool = true | false
oper = + | -

app = atomic
    | app atomic

atomic = int
       | var
       | bool
       | ( expr )
       | let val var = expr in expr end
```

Introducing Precedence

```
expr =                                          bool = true | false
                                                oper = + | -


infix = app
      | infix oper infix

app = atomic
    | app atomic

atomic = int
       | var
       | bool
       | ( expr )
       | let val var = expr in expr end
```

Introducing Precedence

```
expr = infix
     | fn var => expr
     | if expr then expr else expr

infix = app
      | infix oper infix

app = atomic
    | app atomic

atomic = int
       | var
       | bool
       | ( expr )
       | let val var = expr in expr end
```

```
bool = true | false
oper = + | -
```

Parsing Strategies

Parsing Strategies

- Two styles:

Parsing Strategies

- Two styles:
  - Top-down parsing: builds tree from the root

Parsing Strategies

- Two styles:
  - Top-down parsing: builds tree from the root
  - Bottom-up parsing: builds tree from the leaves

Parsing Strategies

- Two styles:
  - Top-down parsing: builds tree from the root
  - Bottom-up parsing: builds tree from the leaves
- Top-down is easy to write by hand

Parsing Strategies

- Two styles:
  - Top-down parsing: builds tree from the root
  - Bottom-up parsing: builds tree from the leaves
- Top-down is easy to write by hand
- Bottom-up is not, but it's used by generators

Parsing Strategies

- Two styles:
  - Top-down parsing: builds tree from the root
  - Bottom-up parsing: builds tree from the leaves
- Top-down is easy to write by hand
- Bottom-up is not, but it's used by generators
- Parser generators: YACC, ANTLR, Bison, etc.

Recursive Descent Parser

- The simplest known parsing strategy; amenable to hand-coding

Recursive Descent Parser

- The simplest known parsing strategy; amenable to hand-coding
- Builds the tree top to bottom, from root to leaves, hence **Descent**

Recursive Descent Parser

- The simplest known parsing strategy; amenable to hand-coding
- Builds the tree top to bottom, from root to leaves, hence **Descent**
- Parallels the structure of the grammar

Recursive Descent Parser

- The simplest known parsing strategy; amenable to hand-coding
- Builds the tree top to bottom, from root to leaves, hence **Descent**
- Parallels the structure of the grammar
- Main idea: each grammar production becomes a function

Recursive Descent Parser

- The simplest known parsing strategy; amenable to hand-coding
- Builds the tree top to bottom, from root to leaves, hence **Descent**
- Parallels the structure of the grammar
- Main idea: each grammar production becomes a function
- Recursion in the grammar translates to recursion in the code, hence **Recursive**

Recursive Descent Parser

- The simplest known parsing strategy; amenable to hand-coding
- Builds the tree top to bottom, from root to leaves, hence **Descent**
- Parallels the structure of the grammar
- Main idea: each grammar production becomes a function
- Recursion in the grammar translates to recursion in the code, hence **Recursive**
- Recursion is the main difference compared to regexes; it needs a stack

Recursive Descent Parser

- The simplest known parsing strategy; amenable to hand-coding
- Builds the tree top to bottom, from root to leaves, hence **Descent**
- Parallels the structure of the grammar
- Main idea: each grammar production becomes a function
- Recursion in the grammar translates to recursion in the code, hence **Recursive**
- Recursion is the main difference compared to regexes; it needs a stack
- Very popular, e.g., Clang uses it for C/C++/Obj-C

Recursive Descent Parser

- The simplest known parsing strategy; amenable to hand-coding
- Builds the tree top to bottom, from root to leaves, hence **Descent**
- Parallels the structure of the grammar
- Main idea: each grammar production becomes a function
- Recursion in the grammar translates to recursion in the code, hence **Recursive**
- Recursion is the main difference compared to regexes; it needs a stack
- Very popular, e.g., Clang uses it for C/C++/Obj-C
- **Parser combinators** are an abstraction over this idea

Code

Removing Left-Recursion

- The current grammar has a problem

Removing Left-Recursion

- The current grammar has a problem
- But, it's only a problem for our current parsing strategy; others can easily cope with it

Removing Left-Recursion

- The current grammar has a problem
- But, it's only a problem for our current parsing strategy; others can easily cope with it
- The problem is that some rules are left-recursive, i.e., the rule itself appears as the first symbol on the left

Removing Left-Recursion

- The current grammar has a problem
- But, it's only a problem for our current parsing strategy; others can easily cope with it
- The problem is that some rules are left-recursive, i.e., the rule itself appears as the first symbol on the left
- This is problematic for a recursive descent parser because the structure of function calls follow the structure of rule definitions

Removing Left-Recursion

- The current grammar has a problem
- But, it's only a problem for our current parsing strategy; others can easily cope with it
- The problem is that some rules are left-recursive, i.e., the rule itself appears as the first symbol on the left
- This is problematic for a recursive descent parser because the structure of function calls follow the structure of rule definitions
- That means infinite recursion in the parser, which isn't good

Left-Recursive Grammar

```
expr = infix
     | fn var => expr
     | if expr then expr else expr

infix = app
      | infix oper infix

app = atomic
    | app atomic

atomic = int
       | var
       | bool
       | ( expr )
       | let val var = expr in expr end
```

```
bool = true | false
oper = + | -
```

Left-Recursive Grammar

```
expr = infix
     | fn var => expr
     | if expr then expr else expr

infix = app
      | infix oper infix

app = atomic
    | app atomic

atomic = int
       | var
       | bool
       | ( expr )
       | let val var = expr in expr end
```

```
bool = true | false
oper = + | -
```

Left-Recursive Grammar

```
app = atomic
    | app atomic
```

Left-Recursive Grammar

```
app = atomic
    | atomic atomic
    | atomic atomic atomic
    | atomic atomic atomic atomic
      ...
```

Left-Recursive Grammar

```
app = atomic
    | atomic atomic
    | atomic (atomic atomic)
    | atomic (atomic (atomic atomic))
      ...
```

Left-Recursive Grammar

```
app = atomic { app }
```

Removing Left-Recursion

```
expr = infix
     | fn var => expr
     | if expr then expr else expr

infix = app
      | infix oper infix

app = atomic { app }

atomic = int
       | var
       | bool
       | ( expr )
       | let val var = expr in expr end
```

```
bool = true | false
oper = + | -
```

Removing Left-Recursion

```
infix = app
      | infix oper infix
```

Removing Left-Recursion

```
infix = app
      | app oper infix
```

Removing Left-Recursion

```
infix = app
      | app oper infix
      | app oper app oper infix
```

Removing Left-Recursion

```
infix = app
      | app oper infix
      | app oper app oper infix
      | app oper app oper app oper infix
```

Removing Left-Recursion

```
infix = app
      | app oper infix
      | app oper app oper infix
      | app oper app oper app oper infix
        ...
```

Removing Left-Recursion

```
infix = app
      | app (oper infix)
      | app (oper app (oper infix))
      | app (oper app (oper app (oper infix)))
        ...
```

Removing Left-Recursion

```
infix = app { oper infix }
```

Removing Left-Recursion

```
expr = infix
     | fn var => expr
     | if expr then expr else expr

infix = app { oper infix }

app = atomic { app }

12 14 13
(12 14) 13

atomic = int
       | var
       | bool
       | ( expr )
       | let val var = expr in expr end
```

```
bool = true | false
oper = + | -
```

github.com / igstan / itake-2016

Homework

- Write a lexer for JSON
- Write a recursive descent parser for JSON
- It's way easier than today's vehicle language
- I promise!
- Specification: json.org

Thank You!

Questions!