Parsing: How Does it Work?

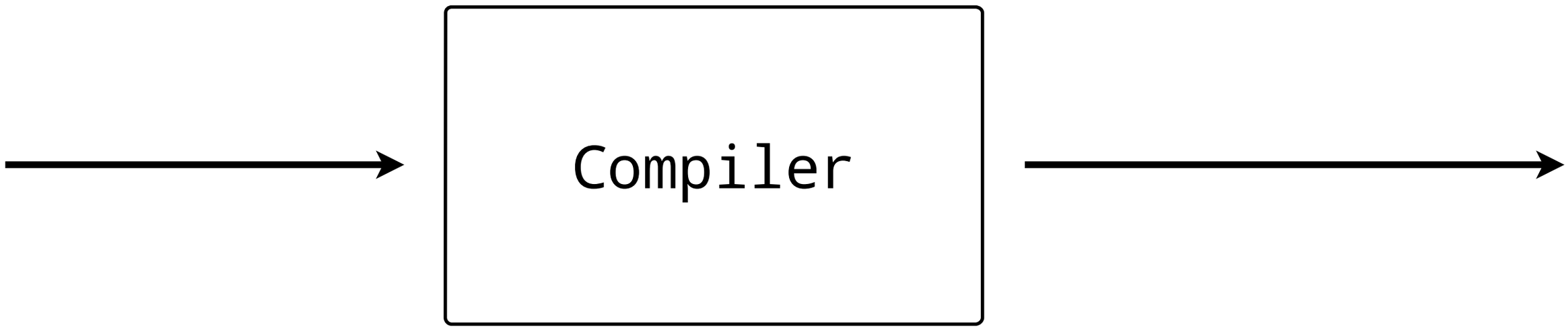
Ionuț G. Stan — Bucharest FP — April 2016

The Plan

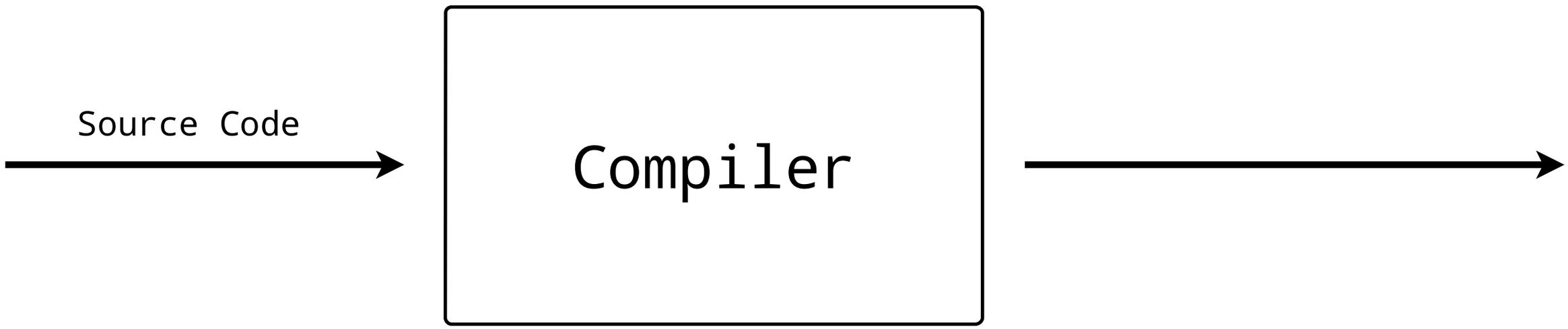
- Compilers Overview
- Vehicle Language — MiniML
- Parsing
  - Lexer
  - Parser

Compilers Overview

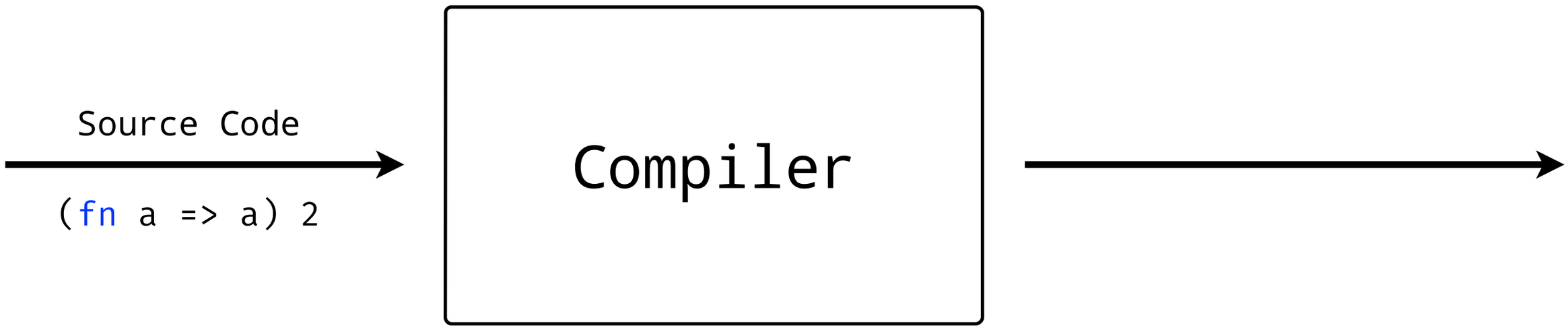
Compilers Overview



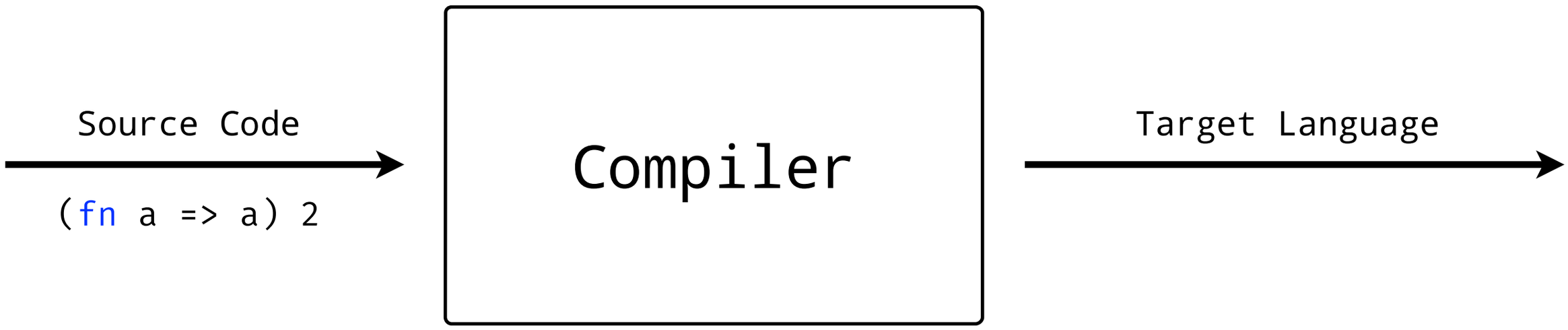
Compilers Overview



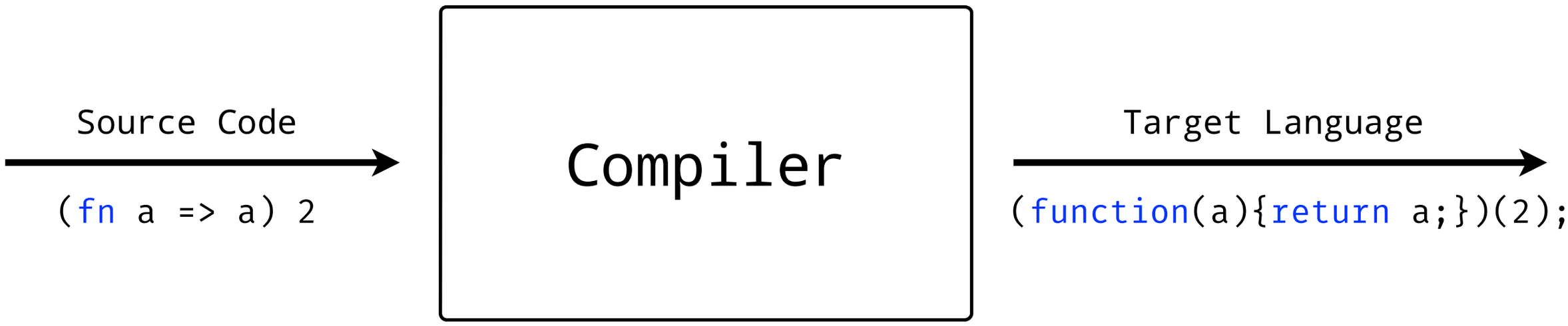
Compilers Overview



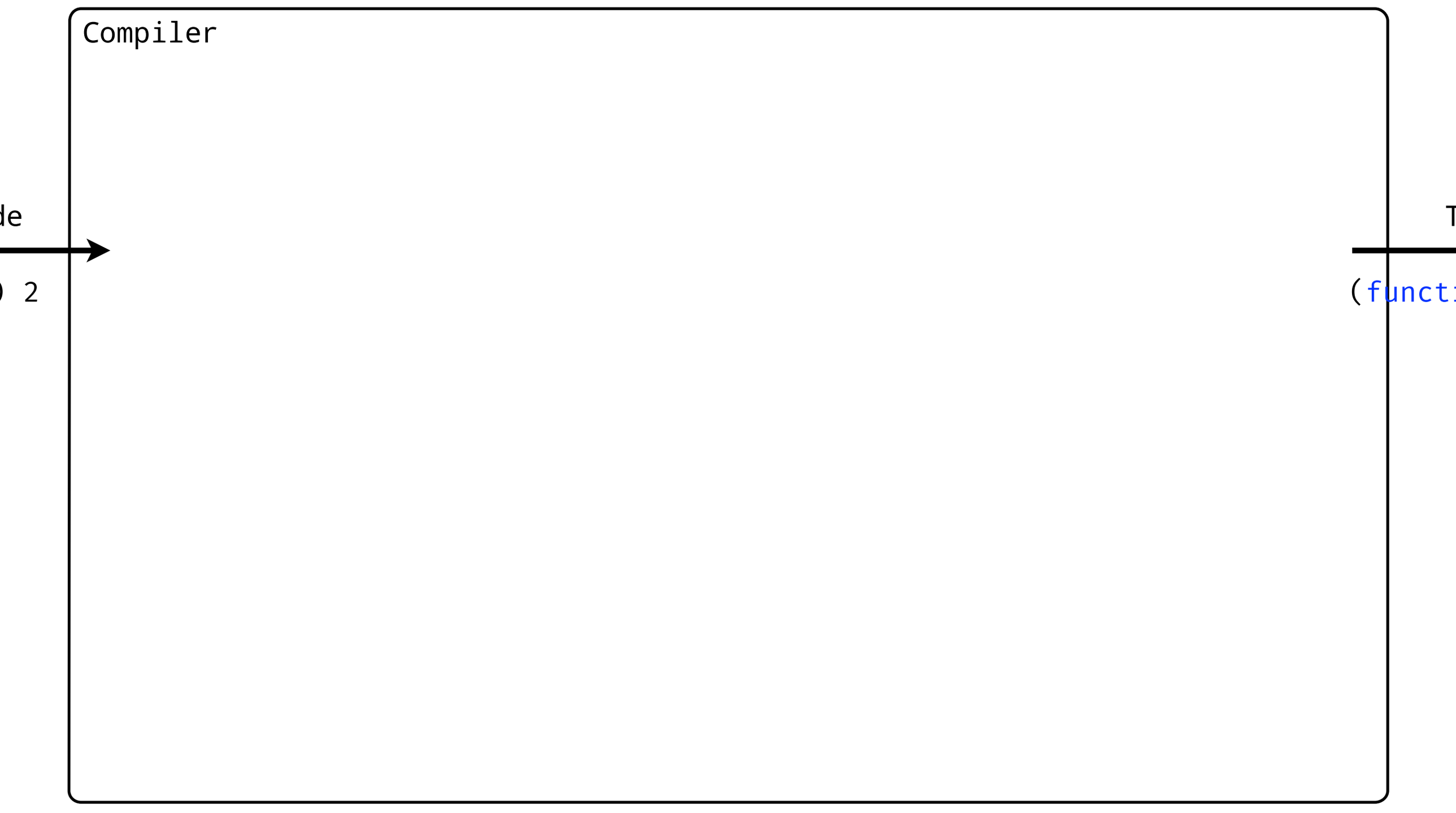
Compilers Overview



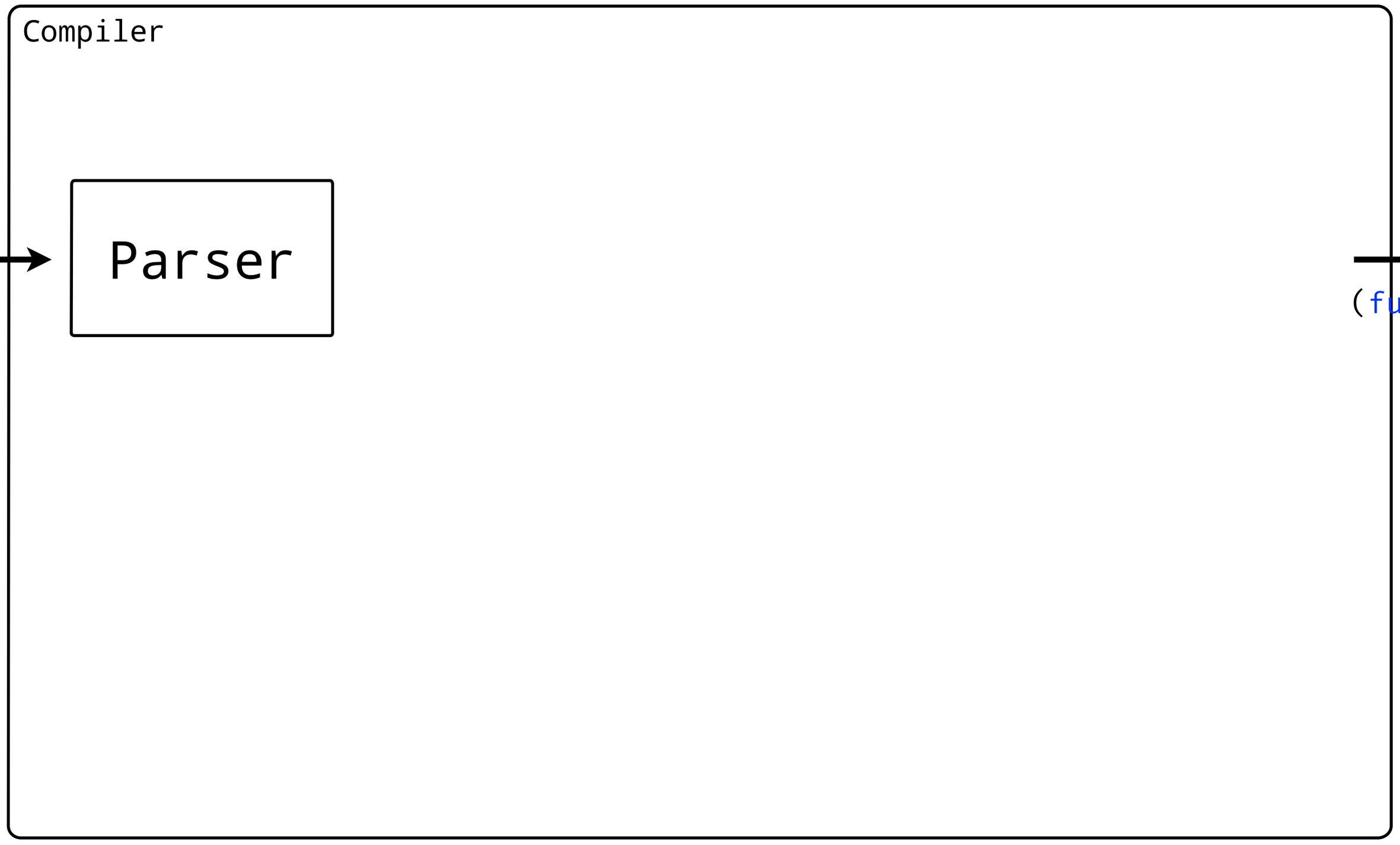
Compilers Overview



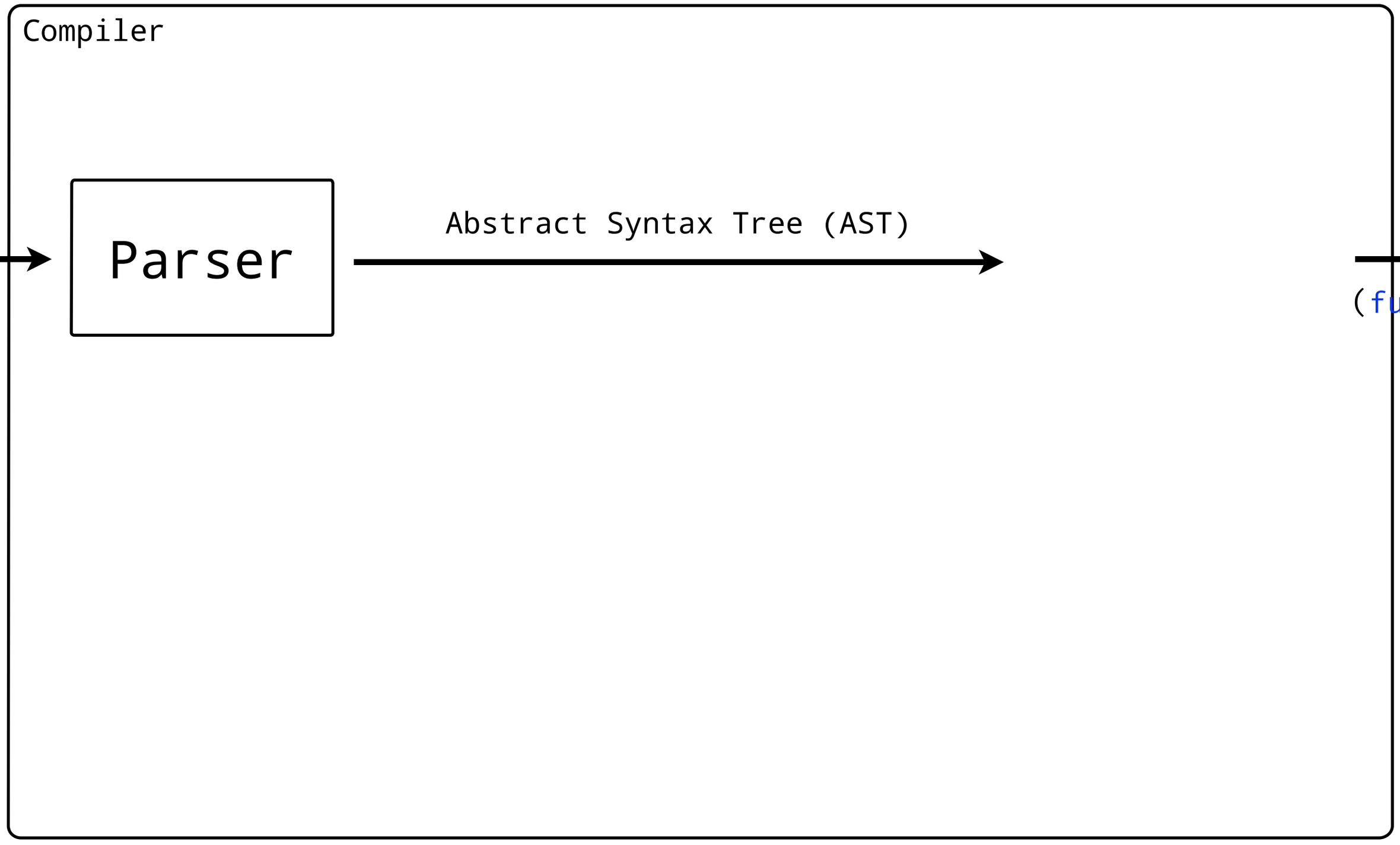
Compilers Overview



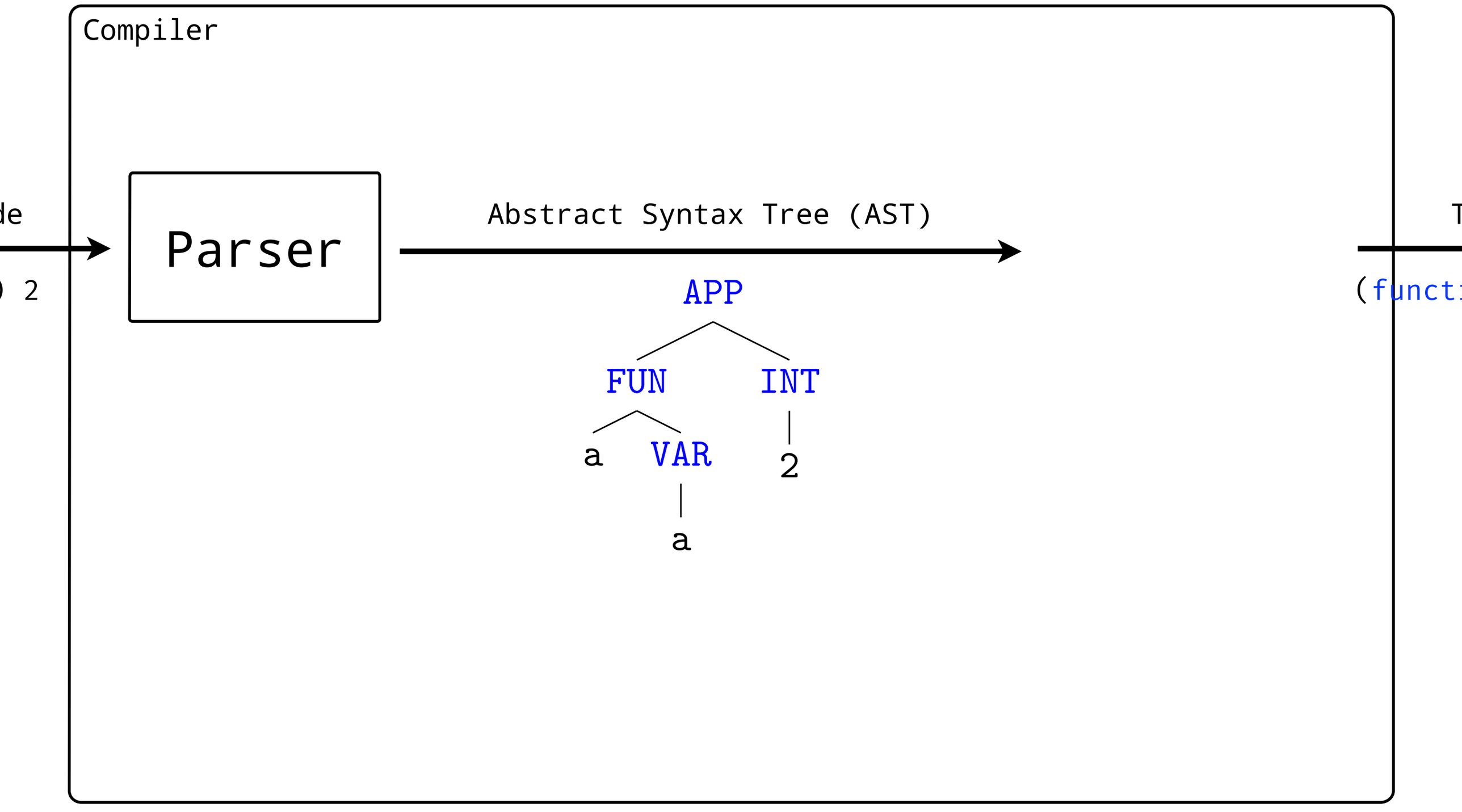
Parsing



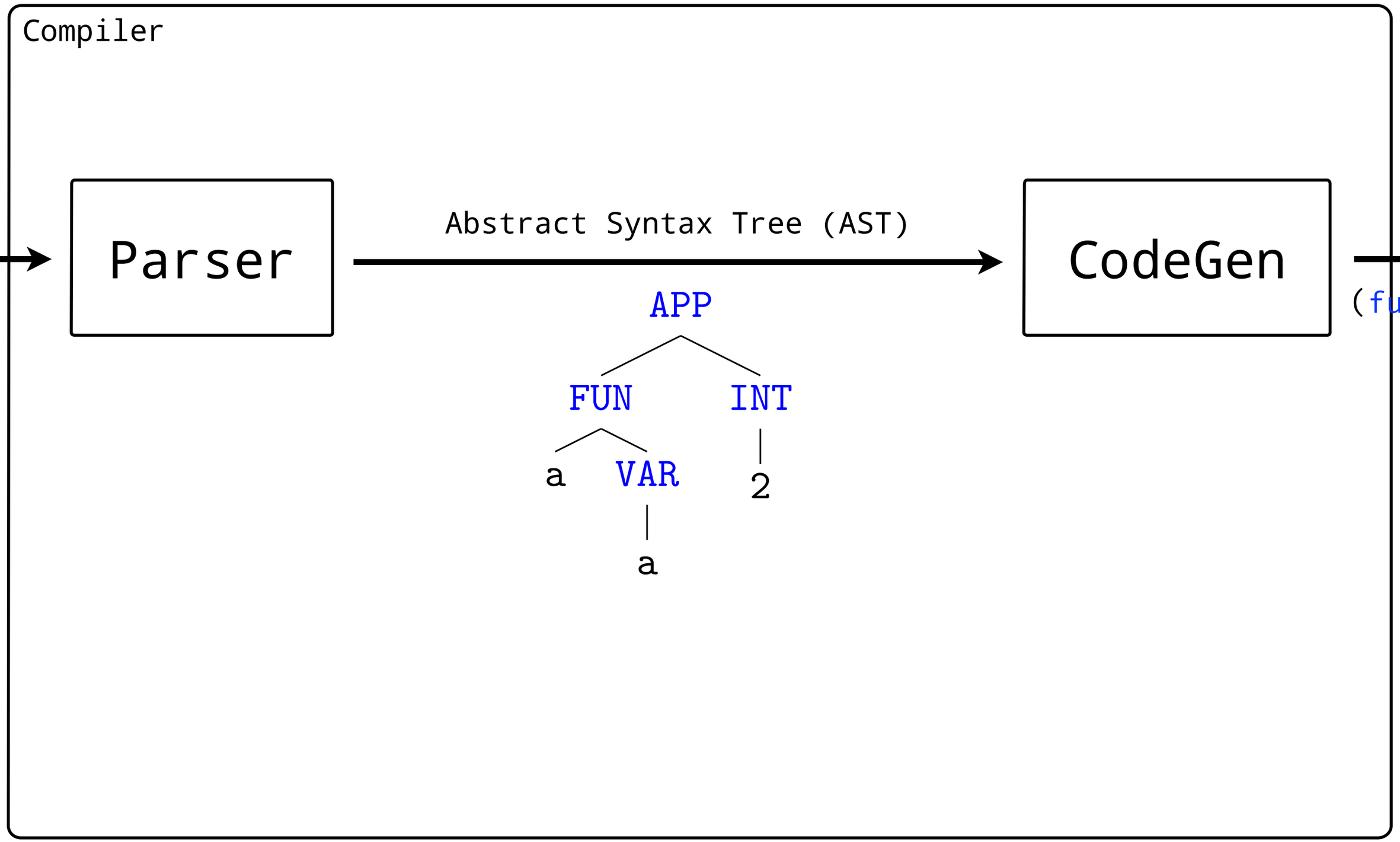
Abstract Syntax Tree



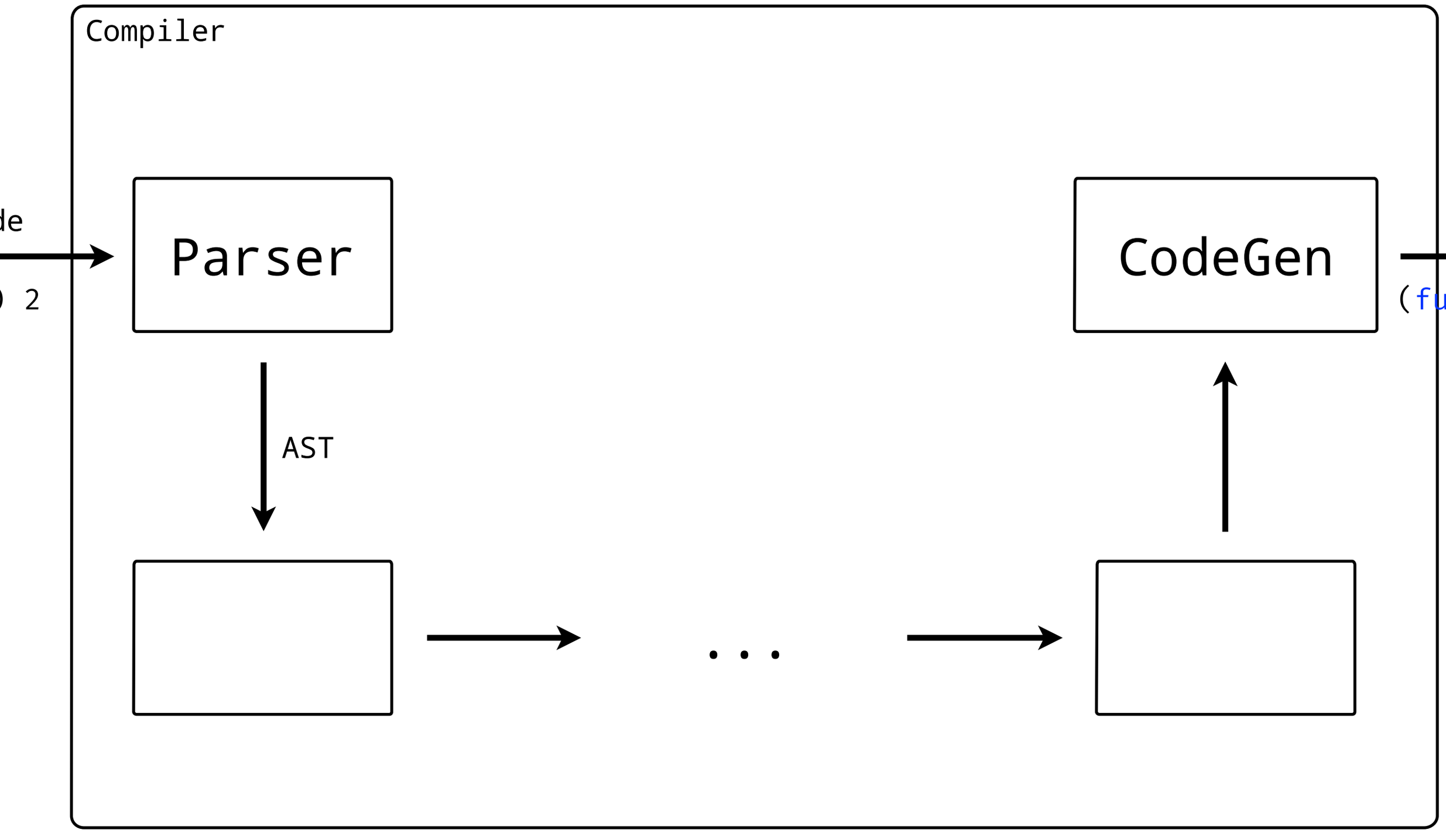
Abstract Syntax Tree



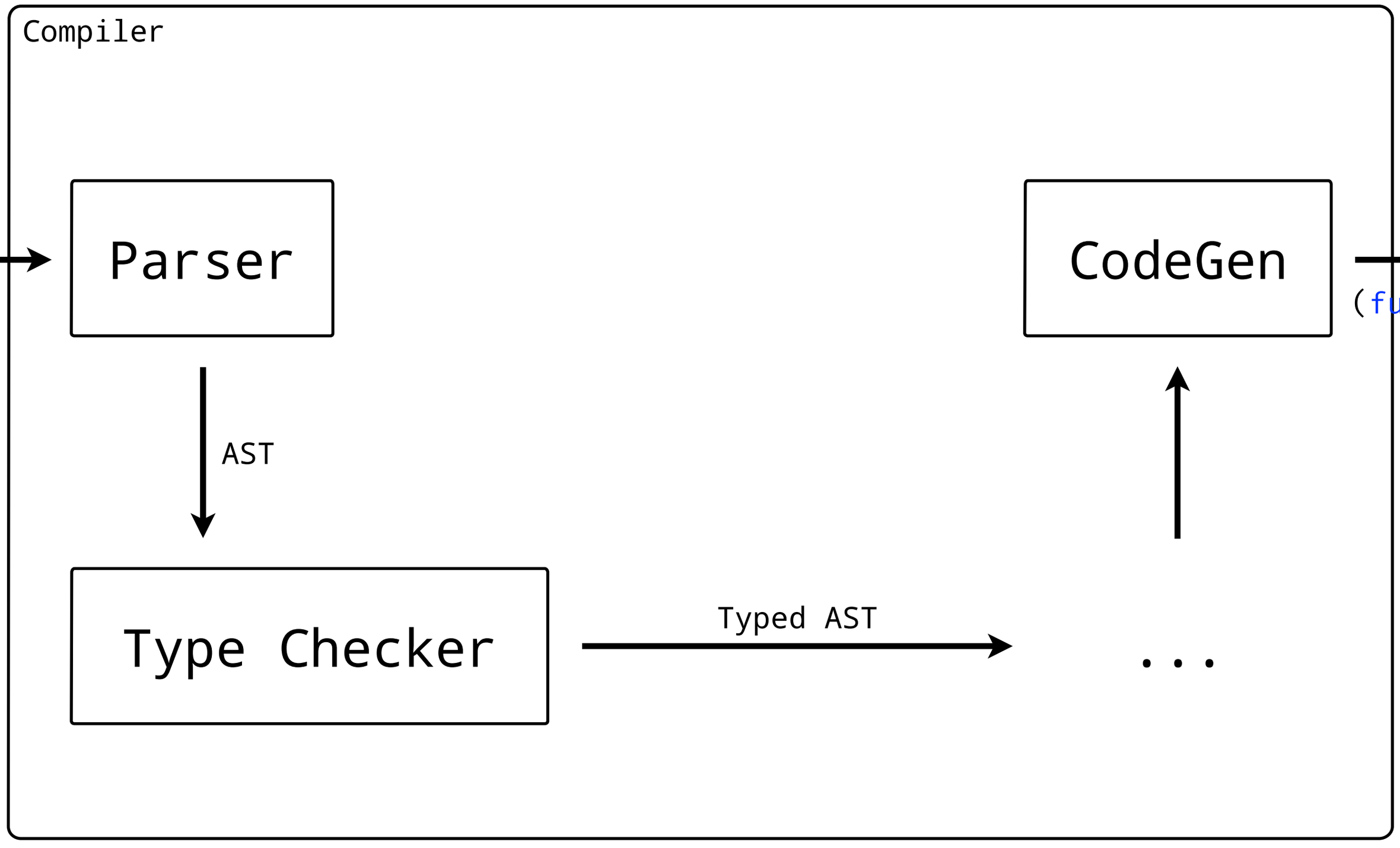
Code Generation



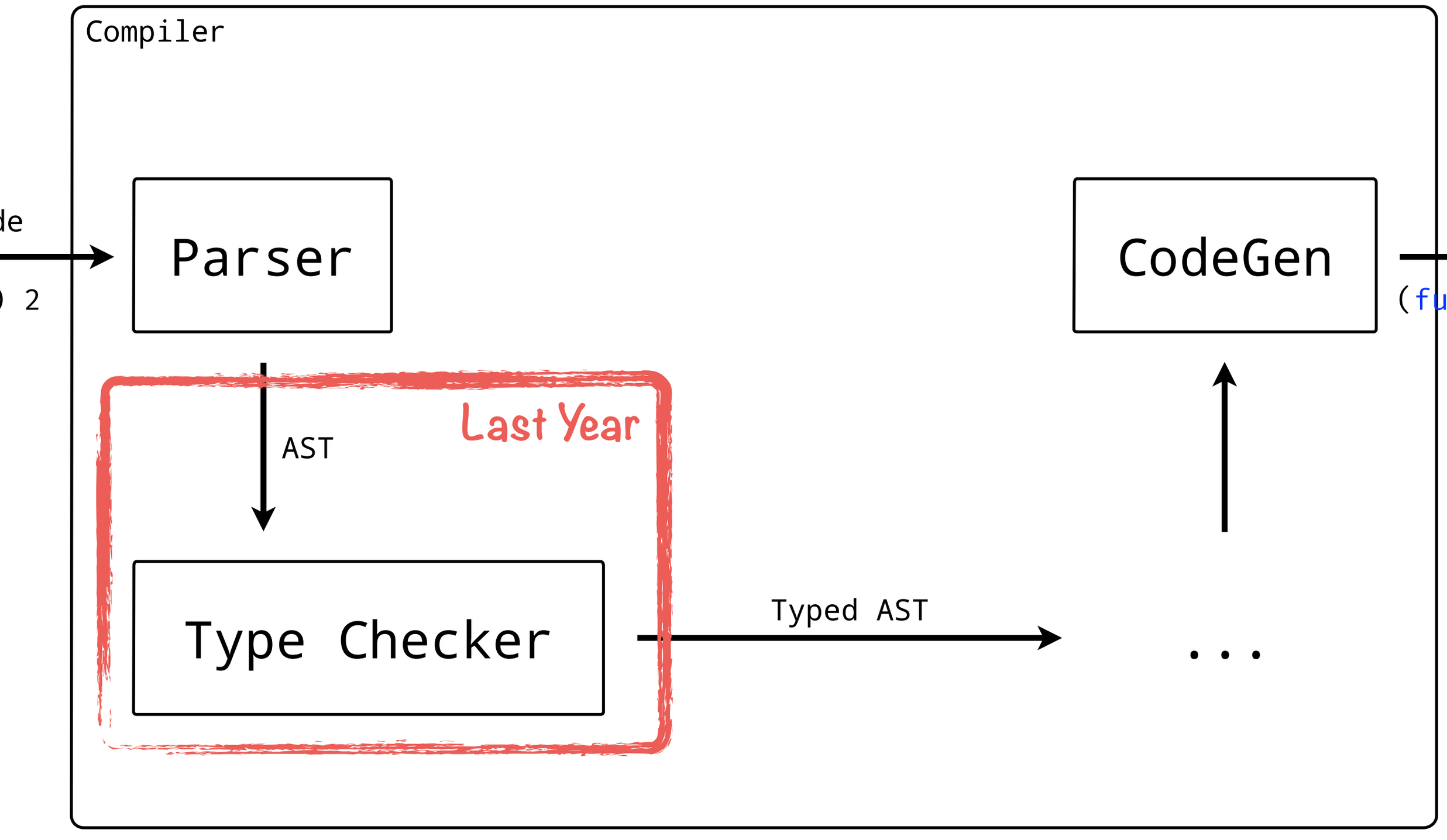
Many Intermediate Phases



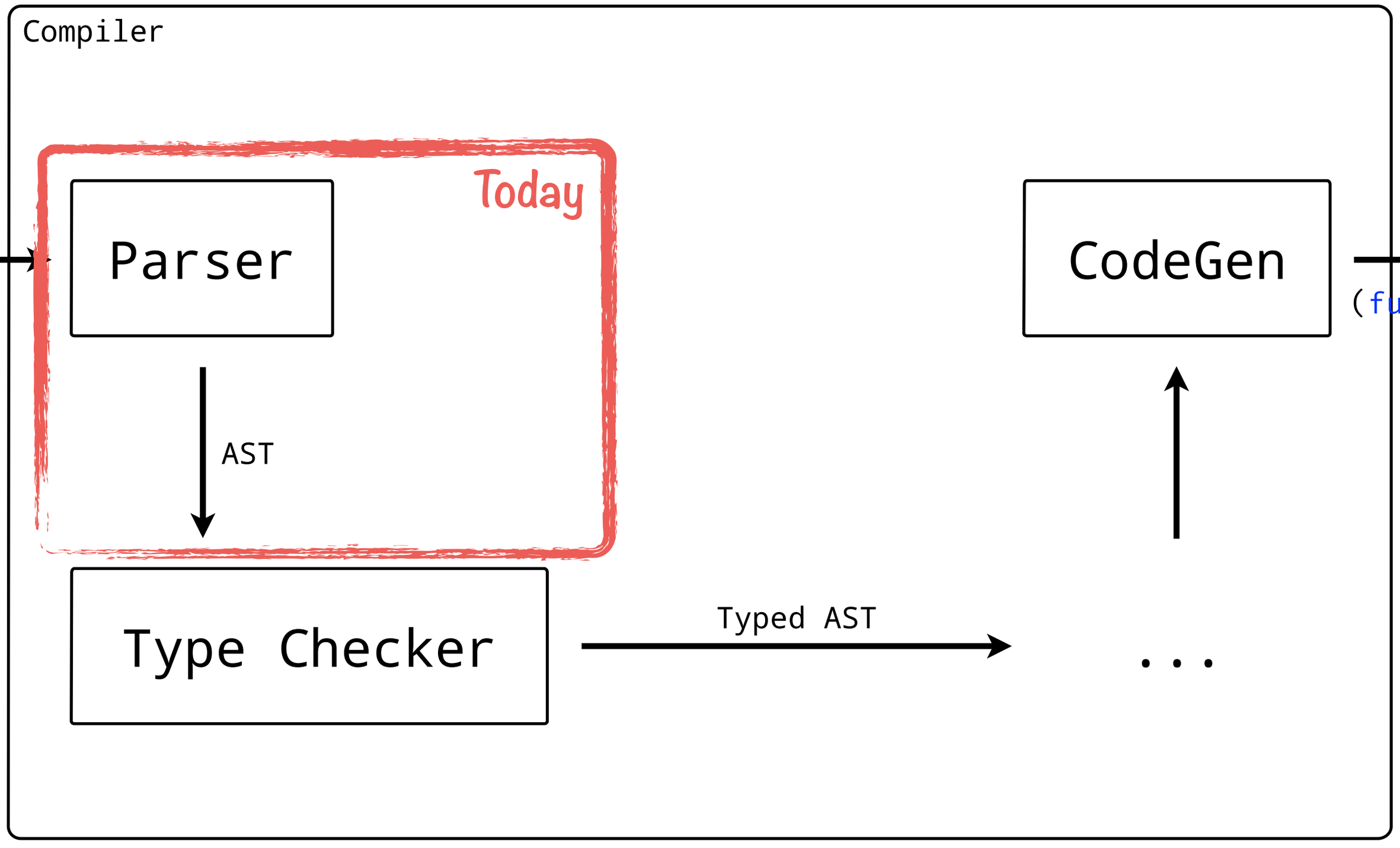
Type Checking



Last Year's Talk



Today's Talk



Vehicle Language

MiniML

MiniML

1. Integers: `1`, `2`, `3`, ...
2. Identifiers (letters only): `foo`, `bar`, `baz`, etc.
3. Booleans: `true` and `false`
4. Anonymous functions (lambdas): `fn a => a`
5. Function application: `inc 42`
6. If expressions: `if cond then t else f`
7. Addition and subtraction: `a + b`, `a - c`
8. Parenthesized expressions: `(expr)`

MiniML

9. Single-binding let blocks:

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

MiniML — Small Example

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

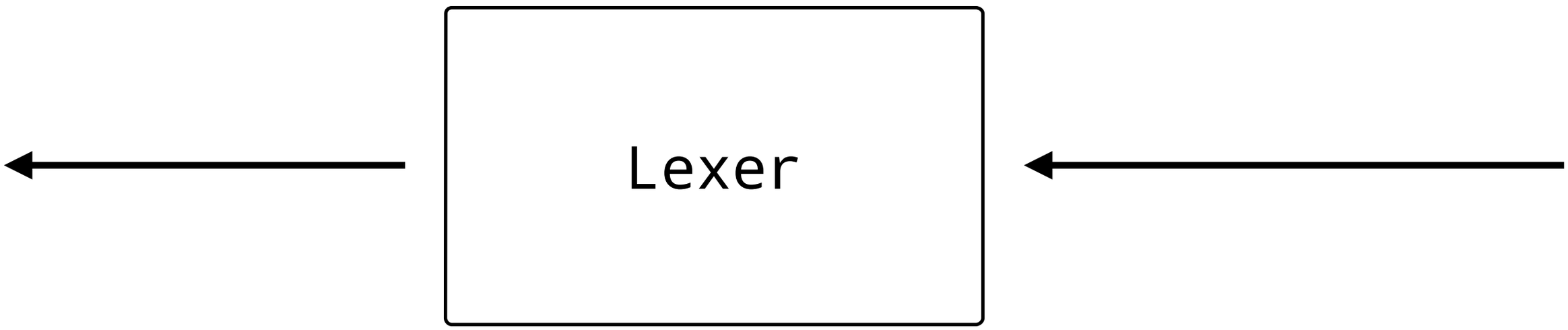
Parsing

Parsing

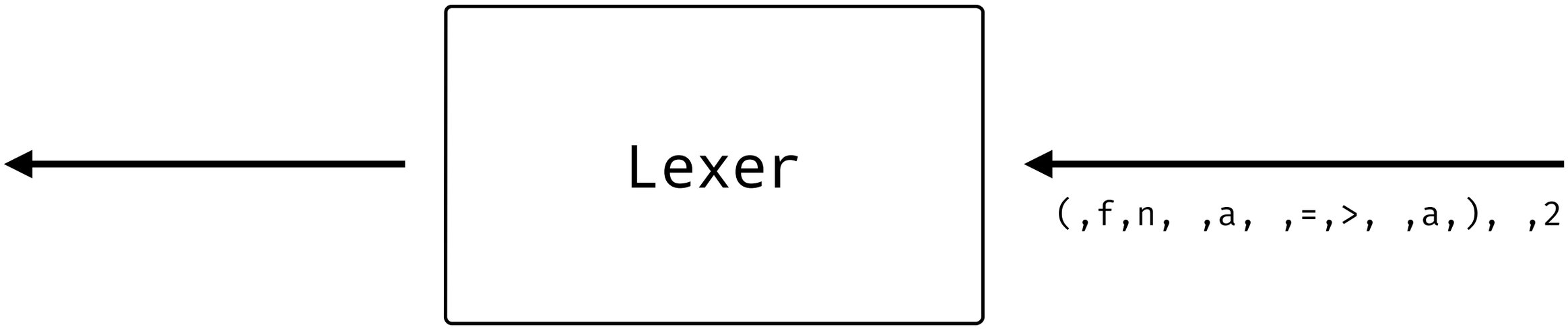
- Purpose: **recover** structure from text
- Traditionally divided in two phases:
  - Lexing: groups **characters** into **words** (**tokens**)
  - Parsing: groups words into **phrases** (AST)
- Other names for lexer: scanner or tokenizer
- Scannerless parsers exist too
- Why lexer + parser then? Mostly efficiency

Lexer

Lexer

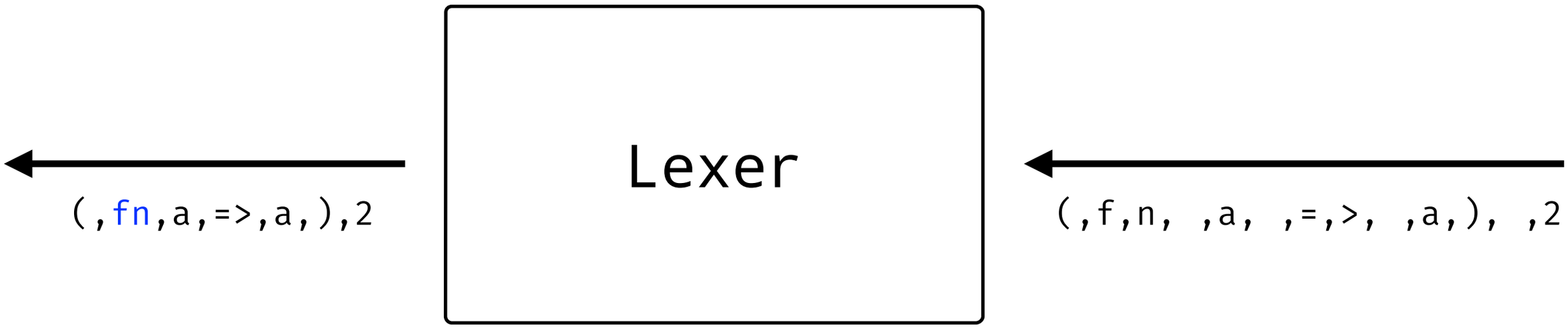


Lexer



- Expects a stream of characters or bytes

Lexer



- Expects a stream of characters or bytes
- Groups them into atomic **semantic** units: **tokens**

Lexer

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

```
val tokens = source.split(" ")
```

Lexer

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

Lexer

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

```
val sum = 1 + 2

val sum=1+2

val str = "spaces matter here"

val str = "spaces /* matter */ here"
```

Lexical Grammar

- Lists the rules for grouping characters into tokens
- Rules specified using regular expressions
- Easy to implement with a RegExp library
- Not extremely difficult without one, either
- Or use a generator, e.g., lex, flex, alex, etc.

MiniML — Lexical Grammar

- integers:
- identifiers:
- symbols:
- keywords:

MiniML — Lexical Grammar

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

MiniML — Lexical Grammar

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

MiniML — Lexical Grammar

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

MiniML — Lexical Grammar

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

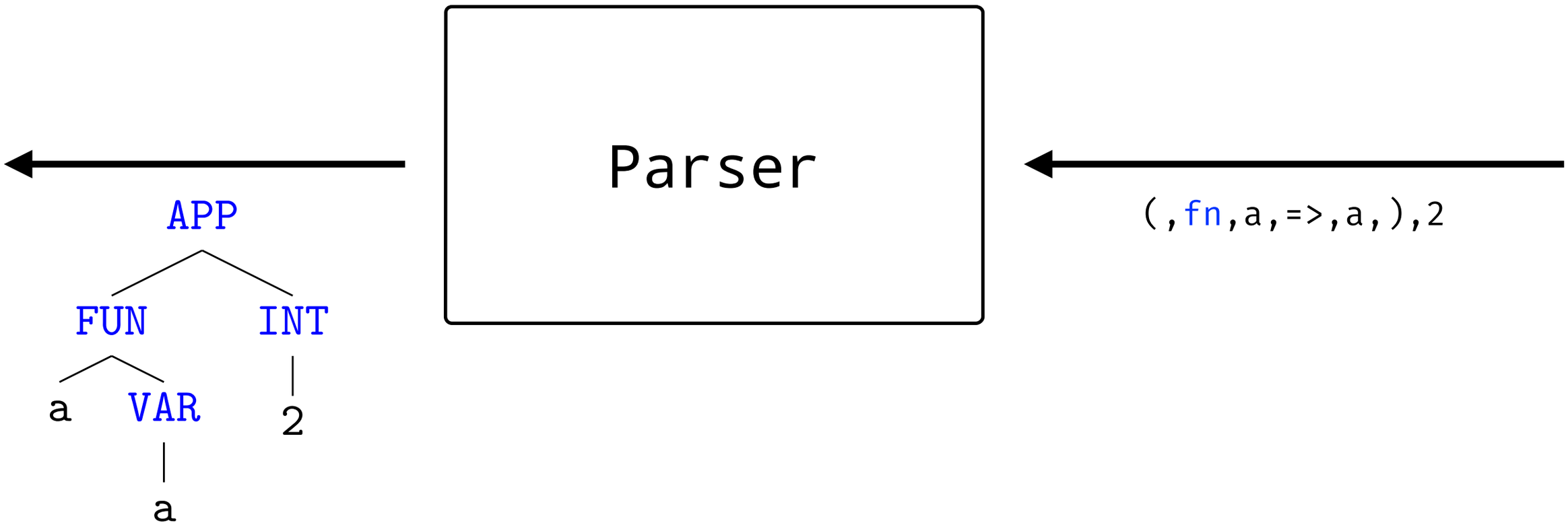
Token Representation

```
sealed trait Token

object Token {
  case class INT(value: Int) extends Token
  case class VAR(value: String) extends Token
  case object IF extends Token
  case object THEN extends Token
  case object ELSE extends Token
  case object FN extends Token
  case object DARROW extends Token
  case object LET extends Token
  case object VAL extends Token
  case object EQUAL extends Token
  case object IN extends Token
  case object END extends Token
  case object LPAREN extends Token
  case object RPAREN extends Token
  case object ADD extends Token
  case object SUB extends Token
  case object TRUE extends Token
  case object FALSE extends Token
}
```

Parser

Parser



Syntactic Grammar

- A grammar tells how tokens can be used together in phrases
- Lexically and syntactically correct: **`val a = 1`**
- Lexically correct, but syntactically incorrect: **`val val val`**
- You need a grammar before writing any code, so…
- Either take it from somewhere, or…
- Write it yourself, or…
- End up with something like PHP

PHP before 5.3

```php
function wat() {
  return array('W', 'A', 'T');
}

echo wat()[0]; // syntax error, unexpected '['
```

MiniML — Syntactic Grammar

```
<EXP> ::=
      <EXP> <EXP> ; function application
    | <EXP> "+" <EXP>
    | <EXP> "-" <EXP>
    | "fn" <VAR> "=>" <EXP>
    | "if" <EXP> "then" <EXP> "else" <EXP>
    | "let" "val" <VAR> "=" <EXP> "in" <EXP> "end"
    | "(" <EXP> ")"
    | <BOOL>
    | <INT>
    | <VAR>
```

AST Representation

```
sealed trait Absyn

object Absyn {
  case class APP(fn: Absyn, arg: Absyn) extends Absyn
  case class ADD(a: Absyn, b: Absyn) extends Absyn
  case class SUB(a: Absyn, b: Absyn) extends Absyn
  case class IF(test: Absyn, yes: Absyn, no: Absyn) extends Absyn
  case class FN(param: String, body: Absyn) extends Absyn
  case class LET(binding: String, value: Absyn, body: Absyn) extends Absyn
  case class BOOL(value: Boolean) extends Absyn
  case class INT(value: Int) extends Absyn
  case class VAR(name: String) extends Absyn
}
```

Parsing Strategies (a)

- Two styles:
  - Top-down parsing: builds AST from root
  - Bottom-up parsing: builds AST from 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.

Parsing Strategies (b)

- Today: **recursive descent parser** (top-down style)
- Very popular — e.g., Clang uses it for C/C++/Obj-C)
- Idea: each grammar production becomes a function
- Productions may be mutually recursive; functions too
- This is the main difference compared to regexes
- Parser combinators are an abstraction over this idea

Recursive Descent Parser

```
<braces> ::= <round>
           | <square>
           | ""

<round>  ::= "(" <braces> ")"
<square> ::= "[" <braces> "]"
```

```
def braces() =
  ???

def round() =
  ???

def square() =
  ???
```

Recursive Descent Parser

- Has a few disadvantages:
  - Can't handle left-recursive grammars
  - Can't handle infix expressions very well:
    - precedence
    - associativity

Code

github.com/igstan/bucharestfp-021

Homework!

- Write a lexer for JSON
- Write a recursive descent parser for JSON
- It's easier than today's vehicle language, I promise!
- Specification: json.org
- Should we try a coding dojo for this?

Thank You!

Questions and 🍺!