Administrivia

Project 2 is due **Wednesday by 11:59PM**

Project 3 is due Monday 4/29 by 11:59PM

Final exam review 4/30 during lecture

Final exam on 5/08 3–5PM in STO B50

Compilation

Principles of Programming Languages
Lecture 24

CAS CS 320

Objectives

Discuss **continuation passing** as an alternative implementation of lexically scoped (immutable) variable bindings

Examine the notion of **compilation**, particularly how it differs from interpretation

Look at examples of compilation from **project 3**

Practice Problem

```
(fun x x -> x)
(fun x x -> x)
(fun x x -> x)
"two" f (f 2)
```

Consider the following OCaml expression. Given ***f*** *is defined previously, what is the type of* ***f****?*

demo

(answer)

Another Practice Problem

```python
def f():
    x = 0
    print(x)

def g():
    print(x)

x = 1
f()
g()
```

`Python`

```
(f):
  0 ▷ x
  x .
; ▷ f

(g): x . ; ▷ g

1 ▷ x
f #
g #
```

`Our Language`

What does this print under dynamic scoping?
under lexical scoping?

Answer

Dynamic:

```
0
0
```

Lexical:

```
0
1
```

```
(f):
  0 ▷ x
  x .
; ▷ f

(g): x . ; ▷ g

1 ▷ x
f #
g #
```

Continuation Passing

Recall: Lexical Scoping

```python
x = 0
def f():
    x = 1
    return(x)
assert(f() == 1)
assert(x == 0)
```

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

Recall: Lexical Scoping

```python
x = 0
def f():
    x = 1
    return(x)
assert(f() == 1)
assert(x == 0)
```

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

Lexical scoping refers the use of textual delimiters to define the scope of a binding

Recall: Lexical Scoping

```python
x = 0
def f():
    x = 1
    return(x)
assert(f() == 1)
assert(x == 0)
```

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

Lexical scoping refers the use of textual delimiters to define the scope of a binding

A binding may be referred to within the delimited textual area of the code

Recall: Lexical Scoping

```python
x = 0
def f():
    x = 1
    return(x)
assert(f() == 1)
assert(x == 0)
```

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

Lexical scoping refers the use of textual delimiters to define the scope of a binding

A binding may be referred to within the delimited textual area of the code

This is also called static scoping because, in theory, scoping errors can be found before the program is run

Recall: Lexical Scoping

```python
x = 0
def f():
    x = 1
    return(x)
assert(f() == 1)
assert(x == 0)
```

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

Lexical scoping refers the use of textual delimiters to define the scope of a binding

A binding may be referred to within the delimited textual area of the code

This is also called static scoping because, in theory, scoping errors can be found before the program is run

(This is far more common in modern programming languages)

Recall: Restricting Scope

scope of f

```python
x = 0
def f():
    x = 1
    return(x)
assert(f() == 1)
assert(x == 0)
```

(Python)

scope of x

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

Recall: Restricting Scope

```python
x = 0
def f():
    x = 1
    return(x)
assert(f() == 1)
assert(x == 0)
```

scope of f

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

scope of x

(OCaml)

Lexical scoping allows us to restrict the scope of a binding. This tends to happen in two ways:

Recall: Restricting Scope

scope of f

```python
x = 0
def f():
    x = 1
    return(x)
assert(f() == 1)
assert(x == 0)
```

(Python)

scope of x

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

Lexical scoping allows us to restrict the scope of a binding. This tends to happen in two ways:

» The binding defines its own scope
  (e.g. let-bindings)

Recall: Restricting Scope

scope of f

```
x = 0
def f():
    x = 1
    return(x)
assert(f() == 1)
assert(x == 0)
```

(Python)

scope of x

```
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

Lexical scoping allows us to restrict the scope of a binding. This tends to happen in two ways:

» The binding defines its own scope (e.g. let-bindings)

» A subroutine or code block defines a scope (e.g. python function)

Recall: Restricting Scope

```
x = 0
def f():           scope of f
    x = 1
    return(x)
assert(f() == 1)
assert(x == 0)
```

(Python)

```
let x = 0          scope of x
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

Lexical scoping allows us to restrict the scope of a binding. This tends to happen in two ways:

focus of project 3

» The binding defines its own scope (e.g. let-bindings)

» A subroutine or code block defines a scope (e.g. python function)

Toy Language

```
<prog>  ::= { <com> }
<com>   ::= <ident> | ▷ <ident> | ()
          | trace <ident>
<val>   ::= <num> | { <com> }
<ident> ::= ...
<num>   ::= ...
```

```
Example Program:
1 ▷ X
2 ▷ X
{
  10 ▷ Y
  trace X
} ▷ F
3 ▷ X
F()
```

A simple stack-based language with variable bindings and subroutines (without parameters or return values).

We will take a *configuration* to be:

(S, E, T, P) or **ERROR**

Toy Language

```
<prog>  ::= { <com> }
<com>   ::= <ident> | ▷ <ident> | ()
          | trace <ident>
<val>   ::= <num> | { <com> }
<ident> ::= ...
<num>   ::= ...
```

```
Example Program:
1 ▷ X
2 ▷ X
{
  10 ▷ Y
  trace X
} ▷ F
3 ▷ X
F()
```

A simple stack-based language with variable bindings and subroutines (without parameters or return values).

We will take a *configuration* to be:

$$(\underbrace{S}_{\text{stack}}, \overset{\text{environment}}{E}, \underbrace{T}_{\text{trace}}, \underbrace{P}_{\text{program}}) \text{ or } \textbf{ERROR}$$

Example

```
1 ▷ X
{ trace X  2 ▷ X } ▷ F
3 ▷ X
{ trace X } ▷ G
F()
G()
```

Example

```
1 ▷ X
{ trace X  2 ▷ X } ▷ F
3 ▷ X
{ trace X } ▷ G
F()
G()
```

What should the trace be after evaluation?

Example

```
1 ▷ X
{ trace X  2 ▷ X } ▷ F
3 ▷ X
{ trace X } ▷ G
F()
G()
```

What should the trace be after evaluation?

<u>Dynamic scoping:</u> ["2", "3"]

Example

```
1 ▷ X
{ trace X  2 ▷ X } ▷ F
3 ▷ X
{ trace X } ▷ G
F()
G()
```

What should the trace be after evaluation?

Dynamic scoping: ["2", "3"]

Lexical scoping: depends

Example

```
x = 1
def f ():
  print(x)
  x = 2
x = 3
def g ():
  print(x)
f()
g()
```

Python

```
let x = 1
let f () =
  let _ = print_int x in
  let x = 2 in
  ()
let x = 3
let g () = print_int x
let _ = f ()
let _ = g ()
```

OCaml

Example

```python
x = 1
def f ():
  print(x)
  x = 2
x = 3
def g ():
  print(x)
f()
g()
```

Python

```ocaml
let x = 1
let f () =
  let _ = print_int x in
  let x = 2 in
  ()
let x = 3
let g () = print_int x
let _ = f ()
let _ = g ()
```

OCaml

It depends on whether we want to interpret it more like a python program or like an OCaml program.

Example

```
x = 1
def f ():
  print(x)
  x = 2
x = 3
def g ():
  print(x)
f()
g()
```

Python

```
let x = 1
let f () =
  let _ = print_int x in
  let x = 2 in
  ()
let x = 3
let g () = print_int x
let _ = f ()
let _ = g ()
```

OCaml

It depends on whether we want to interpret it more like a python program or like an OCaml program.

In the first case, variables in scope are **mutable.** This required maintaining a call stack with local variables for each function call.

Example

```python
x = 1
def f ():
  print(x)
  x = 2
x = 3
def g ():
  print(x)
f()
g()
```

Python

```ocaml
let x = 1
let f () =
  let _ = print_int x in
  let x = 2 in
  ()
let x = 3
let g () = print_int x
let _ = f ()
let _ = g ()
```

OCaml

It depends on whether we want to interpret it more like a python program or like an OCaml program.

In the first case, variables in scope are **mutable.** This required maintaining a call stack with local variables for each function call.

In the second, variables in scope are **immutable.** *We can use a simpler semantics.*

Recall: Closures

```
(Env, P, ...)
```

Recall: Closures

`(Env, P, ...)`

`A` **`closure`** `is a subroutine together with an environment and other data which may be useful for executing the function (name, pointer to activation record where the function is defined`

Recall: Closures

`(Env, P, ...)`

A **closure** is a subroutine together with an environment and other data which may be useful for executing the function (name, pointer to activation record where the function is defined

Env contains captured bindings, the bindings which were defined they may not exist when the function is called

Closures and Immutable Variables

$$\frac{}{\texttt{(S, E, T, \{ Q \} P)} \longrightarrow \texttt{([E, Q] :: S, E, T, P)}}$$

Closures and Immutable Variables

$$\frac{}{(\mathtt{S},\ \mathtt{E},\ \mathtt{T},\ \{\ \mathtt{Q}\ \}\ \mathtt{P}) \longrightarrow ([\ \mathtt{E},\ \mathtt{Q}\ ] :: \mathtt{S},\ \mathtt{E},\ \mathtt{T},\ \mathtt{P}\ )}$$

`If bindings are` **`immutable`**`, then the bindings available to a function when it is defined are` **`fixed`**

Closures and Immutable Variables

$$\frac{}{\texttt{(S, E, T, \{ Q \} P)} \longrightarrow \texttt{([E, Q] :: S, E, T, P)}}$$

If bindings are **immutable**, then the bindings available to a function when it is defined are **fixed**

When we define a function, the closure remembers the **entire environment**

Closures and Immutable Variables

$$\frac{}{\texttt{(S, E, T, \{ Q \} P)} \longrightarrow \texttt{([E, Q] :: S, E, T, P)}}$$

If bindings are **immutable**, then the bindings available to a function when it is defined are **fixed**

When we define a function, the closure remembers the **entire environment**

If the environment changes, the closure still has its **local copy**

Continuation-Passing Style (Calling)

$$\frac{}{\texttt{([C, Q] :: S, E, () P)} \longrightarrow \texttt{([E, P] :: S, C, Q)}}$$

Continuation-Passing Style (Calling)

$$\frac{}{\texttt{([C, Q] :: S, E, () P)} \longrightarrow \texttt{([E, P] :: S, C, Q)}}$$

`We can also use this mechanism to` **`return`** `from functions.`

Continuation-Passing Style (Calling)

$$\frac{}{\texttt{([C, Q] :: S, E, () P)} \longrightarrow \texttt{([E, P] :: S, C, Q)}}$$

We can also use this mechanism to **return** from functions.

We put a closure on the stack when **calling** a function which describes what state the project should return to after the function is done.

Continuation-Passing Style (Calling)

$$\frac{}{\texttt{([C, Q] :: S, E, () P)} \longrightarrow \texttt{([E, P] :: S, C, Q)}}$$

```
We can also use this mechanism to return from functions.
```

We can also use this mechanism to **return** from functions.

We put a closure on the stack when **calling** a function which describes what state the project should return to after the function is done.

This is called a **continuation**.

Continuation-Passing Style (Calling)

$$\frac{}{(\underbrace{[C,\ Q]}_{\text{callee}} :: S,\ E,\ ()\ P) \longrightarrow ([E,\ P] :: S,\ C,\ Q)}$$

We can also use this mechanism to **return** from functions.

We put a closure on the stack when **calling** a function which describes what state the project should return to after the function is done.

This is called a **continuation**.

Continuation-Passing Style (Calling)

$$\frac{}{(\underbrace{[C,\ Q]}_{\texttt{callee}} :: S,\ E,\ ()\ P) \longrightarrow (\underbrace{[E,\ P]}_{\texttt{current continuation}} :: S,\ C,\ Q)}$$

We can also use this mechanism to **return** from functions.

We put a closure on the stack when **calling** a function which describes what state the project should return to after the function is done.

This is called a **continuation**.

Continuation-Passing Style (Returning)

$$\frac{}{\texttt{([E, P] :: S, C', } \epsilon \texttt{)} \longrightarrow \texttt{(S, E, P)}}$$

Continuation-Passing Style (Returning)

$$\frac{}{\texttt{([E, P] :: S, C', } \epsilon \texttt{)} \longrightarrow \texttt{(S, E, P)}}$$

If we reach the end of the function and the current continuation is on top, we **restore** the environment and program.

Continuation-Passing Style (Returning)

$$\frac{}{\texttt{([E, P] :: S, C', }\epsilon\texttt{)} \longrightarrow \texttt{(S, E, P)}}$$

If we reach the end of the function and the current continuation is on top, we **restore** the environment and program.

(In reality, we would achieve this with a **return statement**)

Continuation-Passing Style (Returning)

$$\frac{}{(\underbrace{\boxed{[E,\ P]}}_{\text{current continuation}}\ ::\ S,\ C',\ \epsilon) \longrightarrow (S,\ E,\ P)}$$

If we reach the end of the function and the current continuation is on top, we **restore** the environment and program.

(In reality, we would achieve this with a **return statement**)

Example (Dynamic Scoping)

Stack:

Program:

```
1 ▷ X
{ trace X  2 ▷ X } ▷ F
3 ▷ X
{ trace X } ▷ G
F()
G()
```

Env:

Trace:

Example (Dynamic Scoping)

Stack:

$\boxed{1}$

Program:

```
▷ X
{ trace X  2 ▷ X } ▷ F
3 ▷ X
{ trace X } ▷ G
F()
G()
```

Env:

Trace:

Example (Dynamic Scoping)

Stack:

Program:

```
{ trace X  2 ▷ X } ▷ F
3 ▷ X
{ trace X } ▷ G
F()
G()
```

Env: X ↦ 1

Trace:

Example (Dynamic Scoping)

Stack:

$\boxed{\texttt{trace X 2 ▷ X}}$

Program:

```
▷ F
3 ▷ X
{ trace X } ▷ G
F()
G()
```

Env: X ↦ 1

Trace:

Example (Dynamic Scoping)

Stack:

Program:

```
3 ▷ X
{ trace X } ▷ G
F()
G()
```

Env: X ↦ 1   F ↦ `trace X  2 ▷ X`

Trace:

Example (Dynamic Scoping)

Stack:

$\boxed{3}$

Program:

```
▷ X
{ trace X } ▷ G
F()
G()
```

Env: X ↦ 1   F ↦ `trace X  2 ▷ X`

Trace:

Example (Dynamic Scoping)

Stack:

Program:

```
{ trace X } ▷ G
F()
G()
```

Env: X ↦ 3   F ↦ `trace X`   `2 ▷ X`

Trace:

Example (Dynamic Scoping)

Stack:

$\boxed{\texttt{trace X}}$

Program:

```
▷ G
F()
G()
```

Env: X ↦ 3 F ↦ trace X 2 ▷ X

Trace:

Example (Dynamic Scoping)

Stack:

Program:

```
F()
G()
```

Env: X ↦ 3   F ↦ trace X 2 ▷ X   G ↦ trace X

Trace:

Example (Dynamic Scoping)

Stack:

$\boxed{\texttt{trace X 2 ▷ X}}$

Program:

```
()
G()
```

Env: X ↦ 3 F ↦ `trace X  2 ▷ X` G ↦ `trace X`

Trace:

Example (Dynamic Scoping)

Stack:

Program:

```
trace X  2 ▷ X
G()
```

Env: X ↦ 3   F ↦ `trace X  2 ▷ X`   G ↦ `trace X`

Trace:

Example (Dynamic Scoping)

Stack:

Program:

```
2 ▷ X
G()
```

Env: X ↦ 3   F ↦ trace X 2 ▷ X   G ↦ trace X

Trace: "3"

Example (Dynamic Scoping)

Stack:

$\boxed{2}$

Program:

```
▷ X
G()
```

Env: X ↦ 3   F ↦ trace X 2 ▷ X   G ↦ trace X

Trace: "3"

Example (Dynamic Scoping)

Stack:

Program:

```
G()
```

Env: X $\mapsto$ 2   F $\mapsto$ trace X 2 ▷ X   G $\mapsto$ trace X

Trace: "3"

Example (Dynamic Scoping)

Stack:

$\boxed{\texttt{trace X}}$

Program:

`()`

Env: `X` $\mapsto$ `2`   `F` $\mapsto$ `trace X  2` $\triangleright$ `X`   `G` $\mapsto$ `trace X`

Trace: "3"

Example (Dynamic Scoping)

Stack:

Program:

```
trace X
```

Env: X ↦ 2   F ↦ trace X 2 ▷ X   G ↦ trace X

Trace: "3"

Example (Dynamic Scoping)

Stack:

Program:

Env: X ↦ 2   F ↦ trace X 2 ▷ X   G ↦ trace X

Trace: "2" "3"

Example (Lexical Scoping via CP)

Stack:

Program:

```
1 ▷ X
{ trace X  2 ▷ X } ▷ F
3 ▷ X
{ trace X } ▷ G
F()
G()
```

Env:

Trace:

Example (Lexical Scoping via CP)

Stack:

$\boxed{1}$

Program:

```
▷ X
{ trace X  2 ▷ X } ▷ F
3 ▷ X
{ trace X } ▷ G
F()
G()
```

Env:

Trace:

Example (Lexical Scoping via CP)

Stack:

Program:

```
{ trace X  2 ▷ X } ▷ F
3 ▷ X
{ trace X } ▷ G
F()
G()
```

Env: X ↦ 1

Trace:

Example (Lexical Scoping via CP)

Stack:

```
[X ↦ 1, trace X  2 ▷ X]
```

Program:

```
▷ F
3 ▷ X
{ trace X } ▷ G
F()
G()
```

Env: X ↦ 1

Trace:

Example (Lexical Scoping via CP)

Stack:

Program:

```
3 ▷ X
{ trace X } ▷ G
F()
G()
```

Env: X ↦ 1   F ↦ [X ↦ 1, trace X 2 ▷ X]

Trace:

Example (Lexical Scoping via CP)

Stack:

$\boxed{3}$

Program:

```
▷ X
{ trace X } ▷ G
F()
G()
```

```
Env: X ↦ 1  F ↦ [X ↦ 1, trace X  2 ▷ X]
```

Trace:

Example (Lexical Scoping via CP)

Stack:

Program:

```
{ trace X } ▷ G
F()
G()
```

Env: X ↦ 3 F ↦ [X ↦ 1, trace X 2 ▷ X]

Trace:

Example (Lexical Scoping via CP)

Stack:

```
[ X ↦ 3  F ↦ ...
, trace X
]
```

Program:

```
▷ G
F()
G()
```

Env: X ↦ 3 F ↦ [X ↦ 1, trace X 2 ▷ X]

Trace:

Example (Lexical Scoping via CP)

Stack:

Program:

```
F()
G()
```

```
              G ↦ [X ↦ 3  F ↦...,  trace X]
Env: X ↦ 3    F ↦ [X ↦ 1, trace X  2 ▷ X]
```

Trace:

Example (Lexical Scoping via CP)

Stack:

```
[X ↦ 1, trace X  2 ▷ X]
```

Program:

```
()
G()
```

```
                   G ↦ [X ↦ 3  F ↦...,  trace X]
  Env: X ↦ 3   F ↦ [X ↦ 1, trace X  2 ▷ X]
Trace:
```

Example (Lexical Scoping via CP)

Stack:

```
[ X ↦ 3  F ↦ ...  G ↦ ...
, G ()
]
```

Program:

```
trace X  2 ▹ X
```

Env: X ↦ 1

Trace:

Example (Lexical Scoping via CP)

Stack:

```
[ X ↦ 3  F ↦ ...  G ↦ ...
, G ()
]
```

Program:

```
2 ▷ X
```

```
  Env: X ↦ 1
Trace: "1"
```

Example (Lexical Scoping via CP)

Stack:

$\boxed{2}$

```
[ X ↦ 3  F ↦ ...  G ↦ ...
, G ()
]
```

Program:

```
▷ X
```

Env: X ↦ 1

Trace: "1"

Example (Lexical Scoping via CP)

Stack:

```
[ X ↦ 3  F ↦ ...  G ↦ ...
, G ()
]
```

Program:

Env: X ↦ 2

Trace: "1"

Example (Lexical Scoping via CP)

Stack:

Program:

```
G ()
```

```
            G ↦ [X ↦ 3  F ↦...,  trace X]
Env: X ↦ 3  F ↦ [X ↦ 1, trace X  2 ▷ X]
Trace: "1"
```

Example (Lexical Scoping via CP)

Stack:

```
[ X ↦ 3  F ↦ ...
, trace X
]
```

Program:

```
()
```

```
                    G ↦ [X ↦ 3  F ↦...,  trace X]
  Env: X ↦ 3  F ↦ [X ↦ 1, trace X  2 ▹ X]
Trace: "1"
```

Example (Lexical Scoping via CP)

Stack:

```
[ X ↦ 3  F ↦ ...  G ↦ ...
, ϵ
]
```

Program:

```
trace X
```

```
Env: X ↦ 3  F ↦ [X ↦ 1, trace X  2 ▹ X]
Trace: "1"
```

Example (Lexical Scoping via CP)

Stack:

```
[ X ↦ 3  F ↦ ...  G ↦ ...
, ϵ
]
```

Program:

Env: X ↦ 3 F ↦ [X ↦ 1, trace X 2 ▷ X]

Trace: "3" "1"

Example (Lexical Scoping via CP)

Stack:

Program:

```
            G ↦ [X ↦ 3  F ↦...,  trace X]
 Env: X ↦ 3  F ↦ [X ↦ 1, trace X  2 ▷ X]
Trace: "3"  "1"
```

Good Practice Problems

Write down the explicitly the operational semantics which make each of the previous examples work.

Write down the example using mutable lexically scoped variables.

Understanding Check

```
1 ▷ X
2 ▷ Y
{
  1 ▷ Y
  trace Y
} ▷ F
F()
trace Y
```

*What is the current continuation pushed to the stack when the function **F** is called?*

Answer

```
[ X ↦ 1  Y ↦ 2
, trace Y
]
```

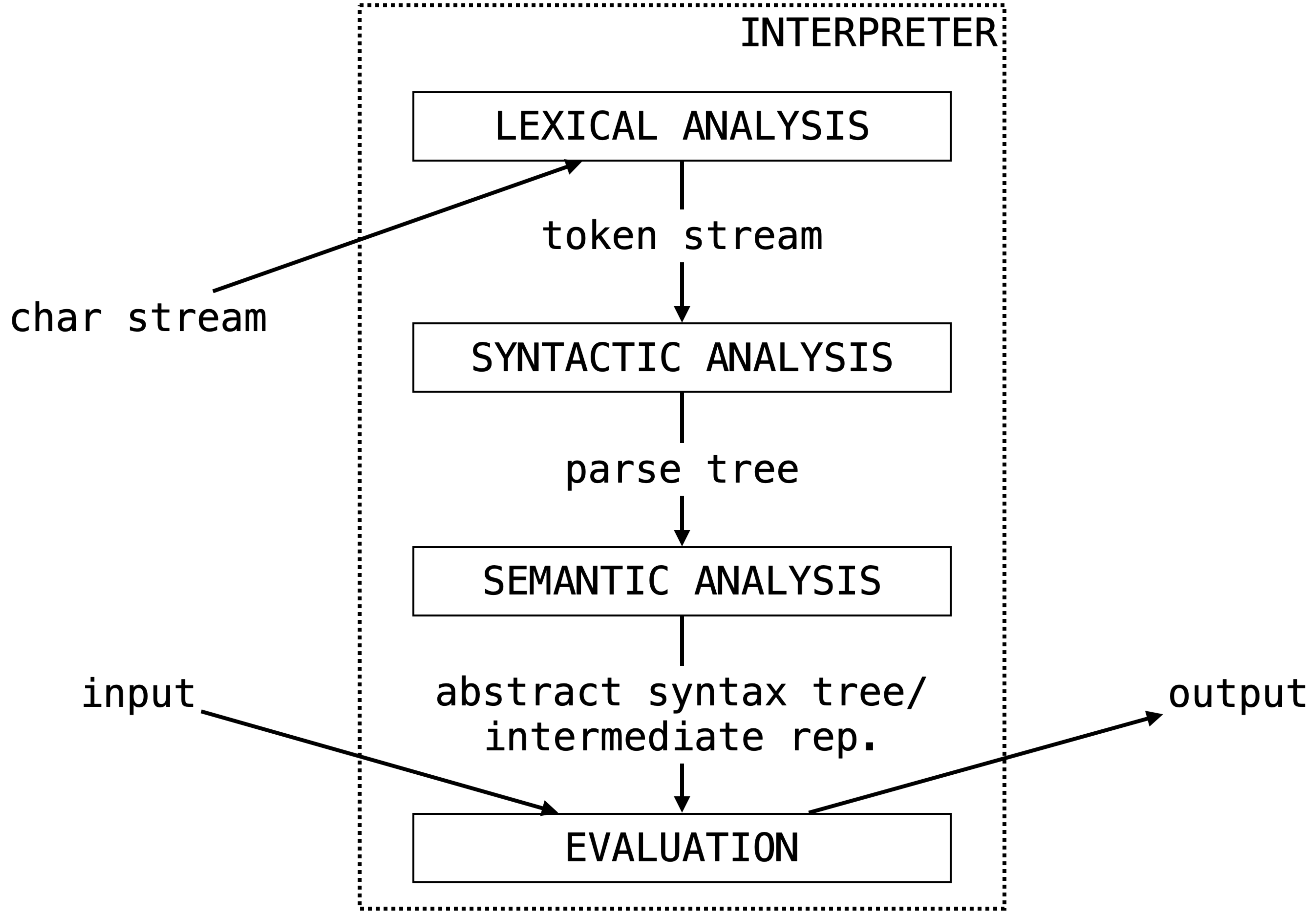
Note that the continuation binds ***Y*** *to the correct value.*

Compilation

Pure Interpretation

So far in this course, we have been looking at **interpretation.**

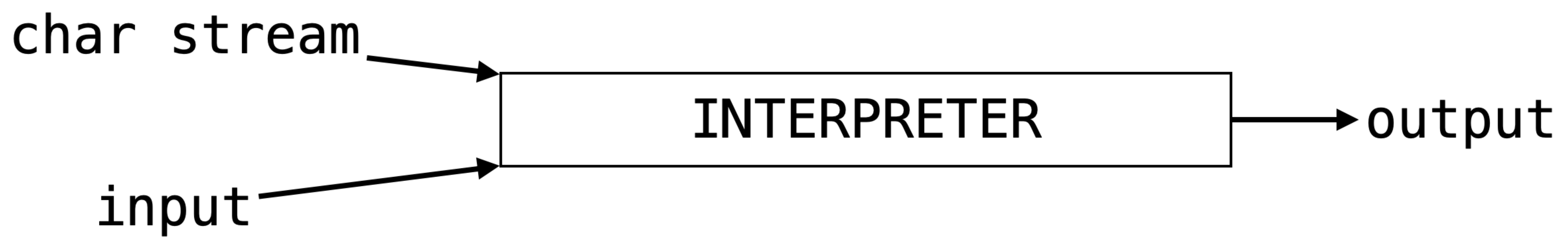
We get a program into a form which we can *immediately* evaluate.

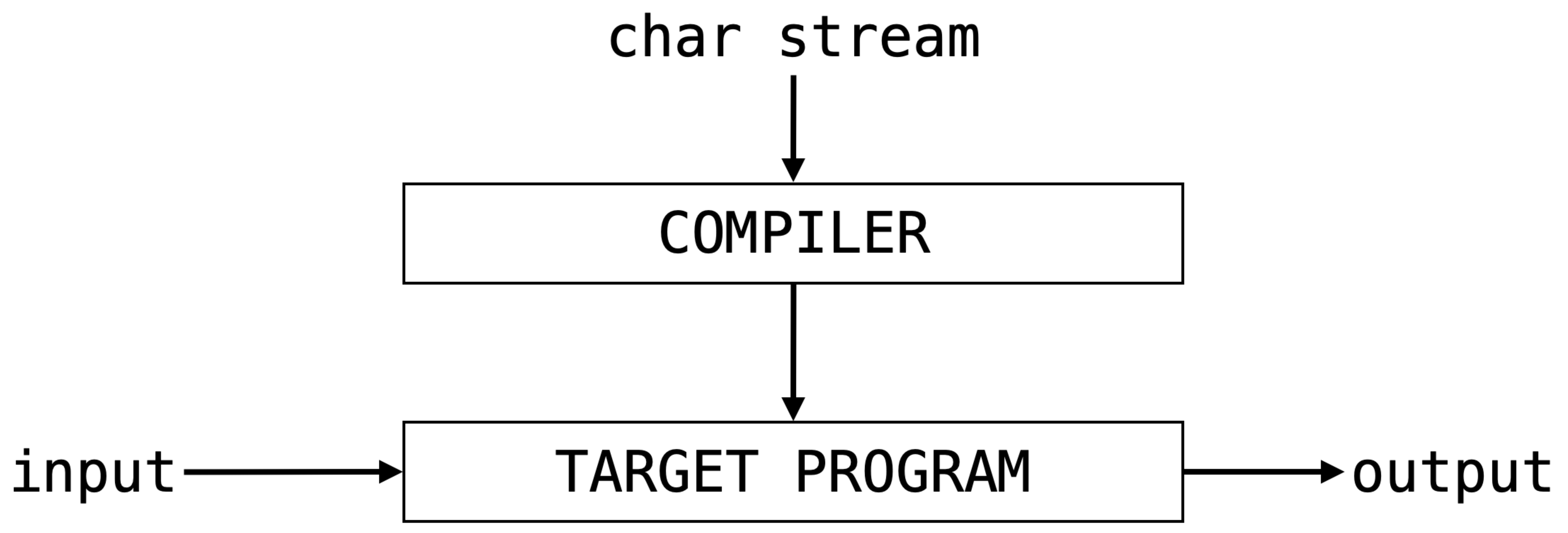


Interpretation vs. Compilation

Compilation is about **translating** a program into one which which can be be interpreted (or assembled) by a *different* program.

Question. *Why would we want to do this?*



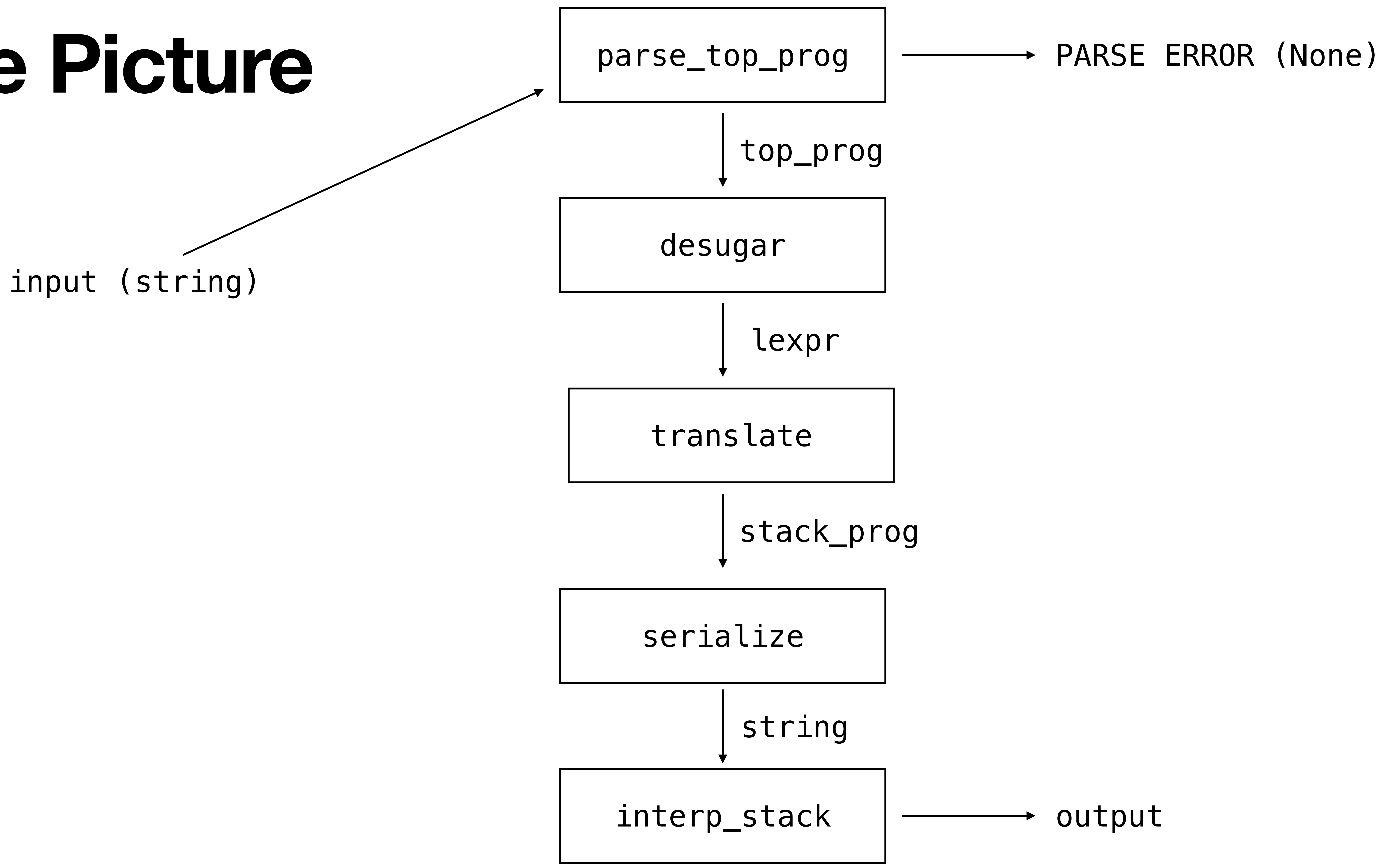


Benefits of Compilation

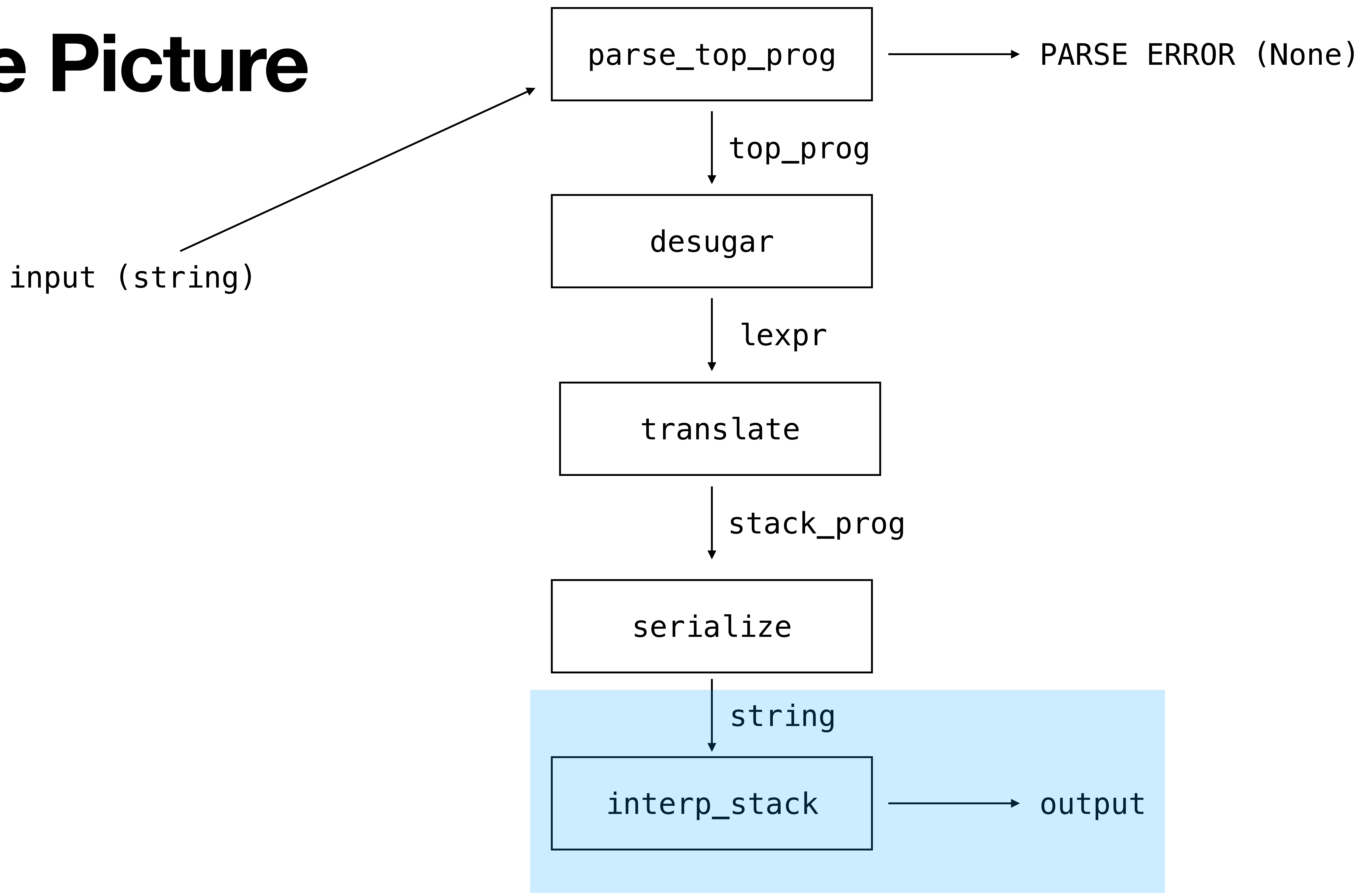
Code Optimization. We can transform the code to eliminate unnecessary parts, or make the structure (e.g., tail-call optimization).

Machine-dependent code generation. We can build our code differently for different machines.

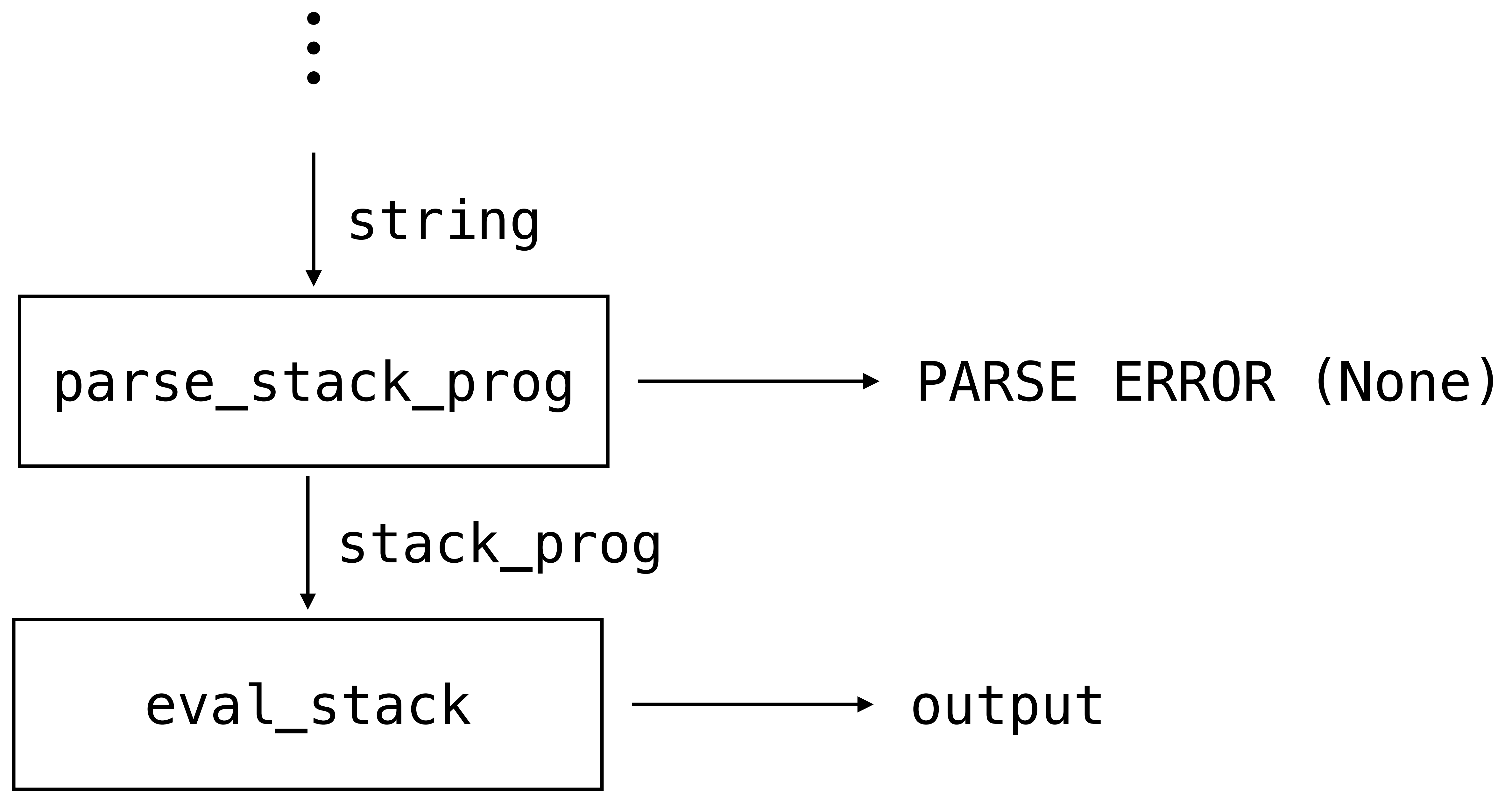
The Picture


input (string)
parse_top_prog
PARSE ERROR (None)
top_prog
desugar
lexpr
translate
stack_prog
serialize
string
interp_stack
output

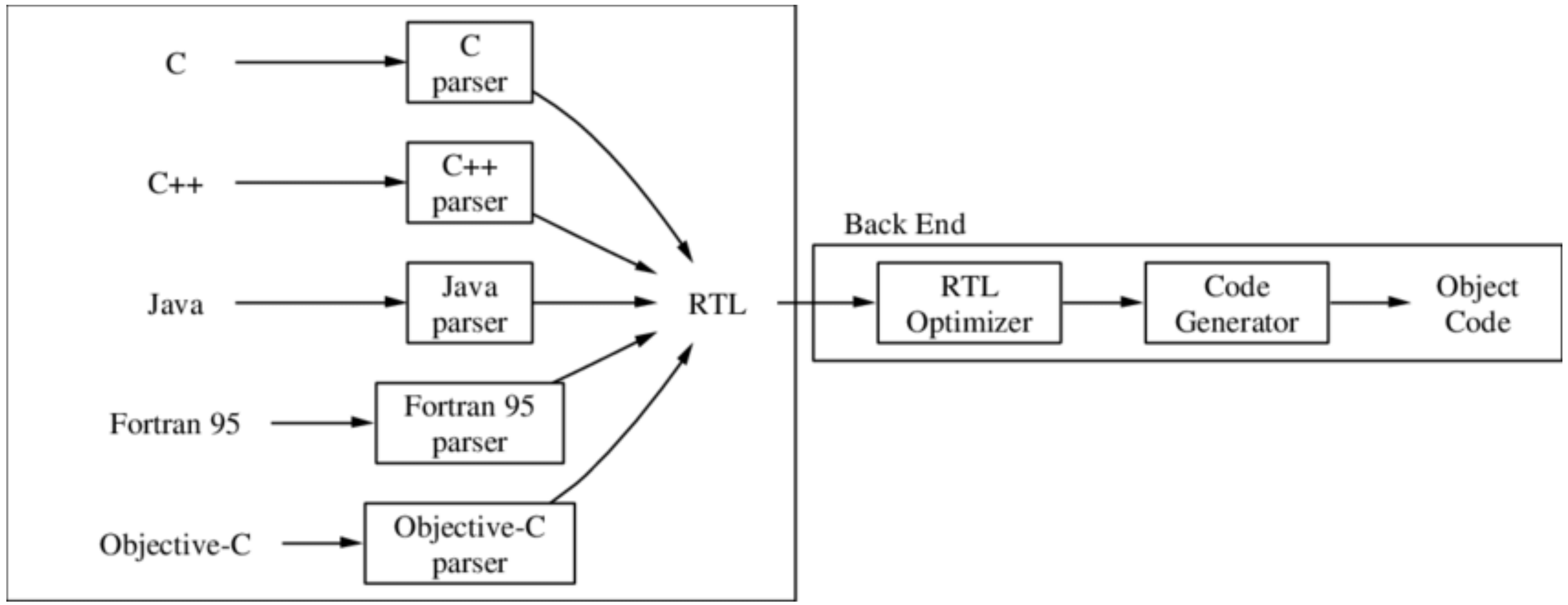
The Picture


parse_top_prog
PARSE ERROR (None)
input (string)
top_prog
desugar
lexpr
translate
stack_prog
serialize
string
interp_stack
output

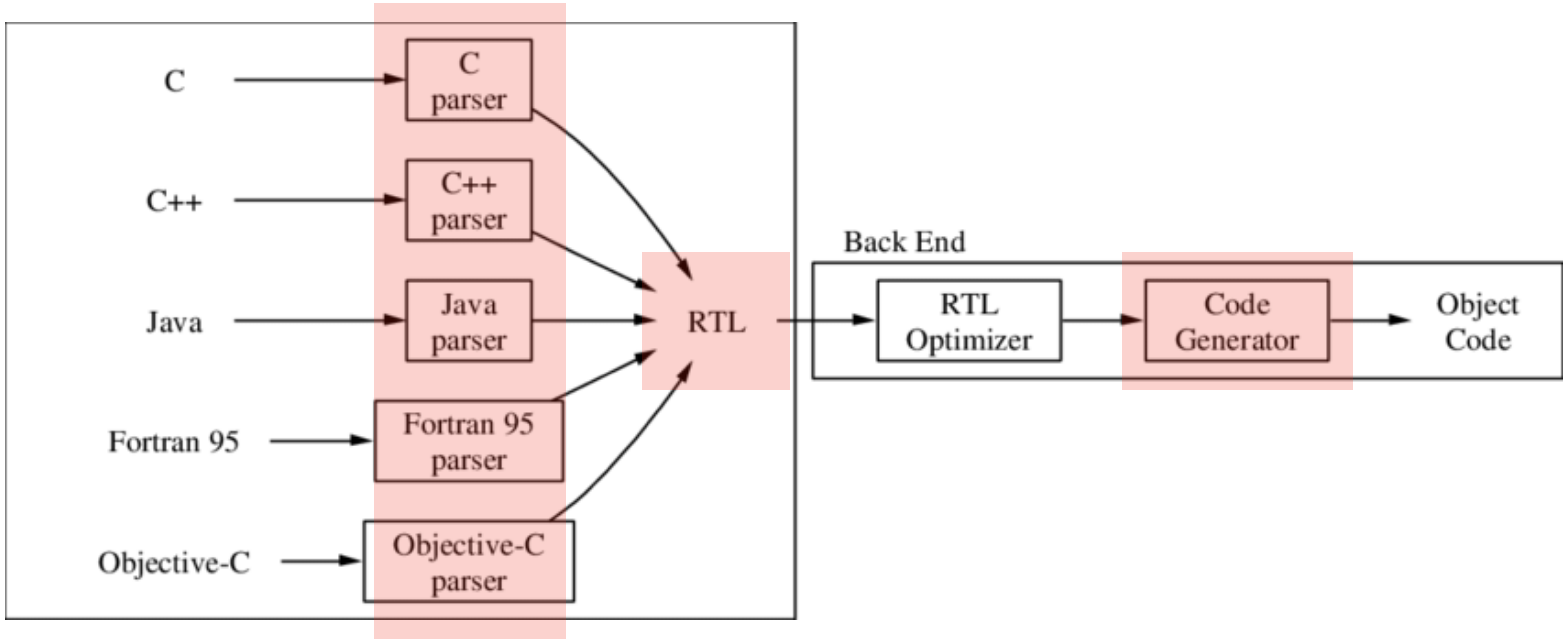
The Picture


string
parse_stack_prog
PARSE ERROR (None)
stack_prog
eval_stack
output

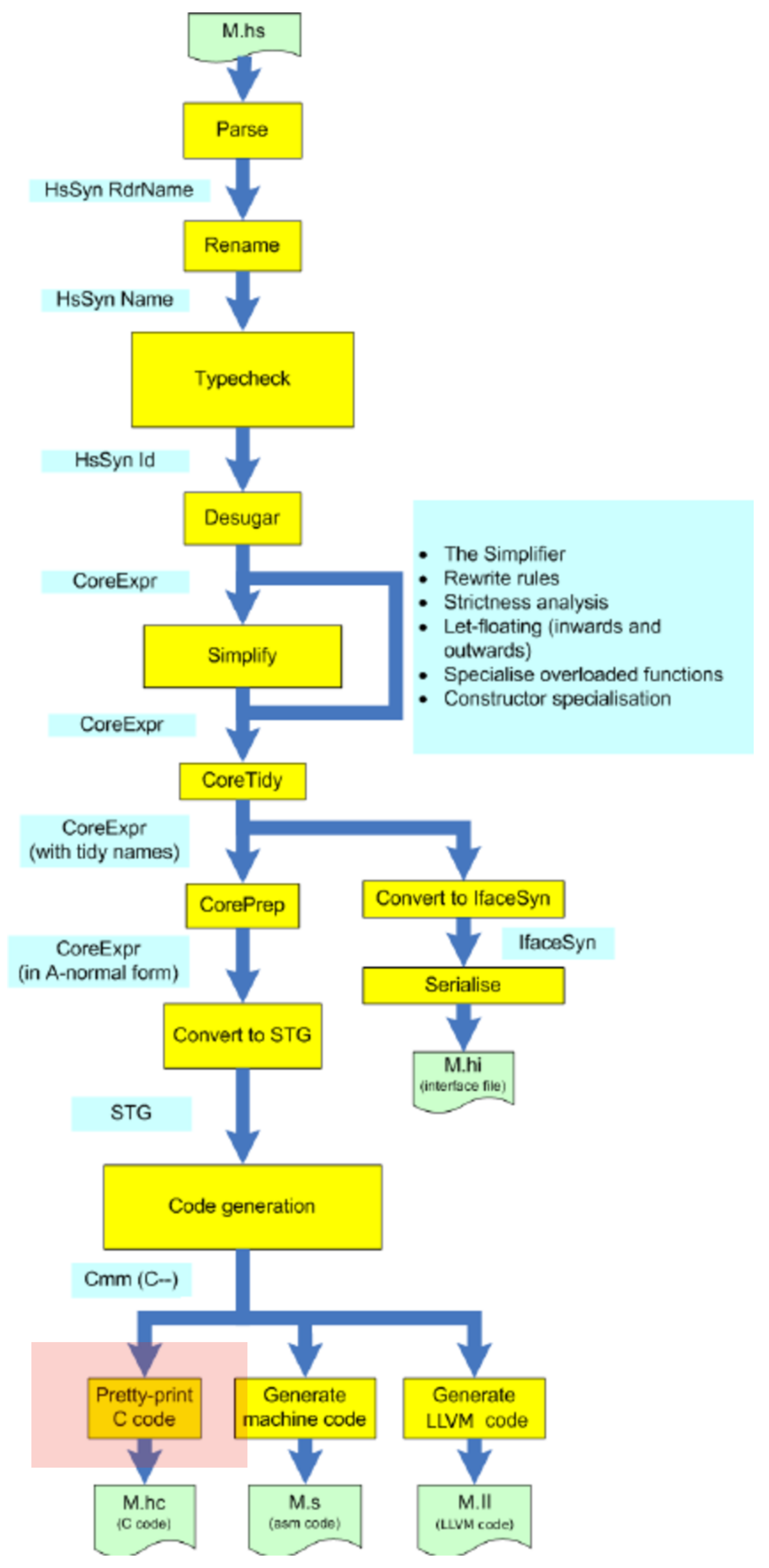
Example Pipelines: gcc



Example Pipelines: gcc

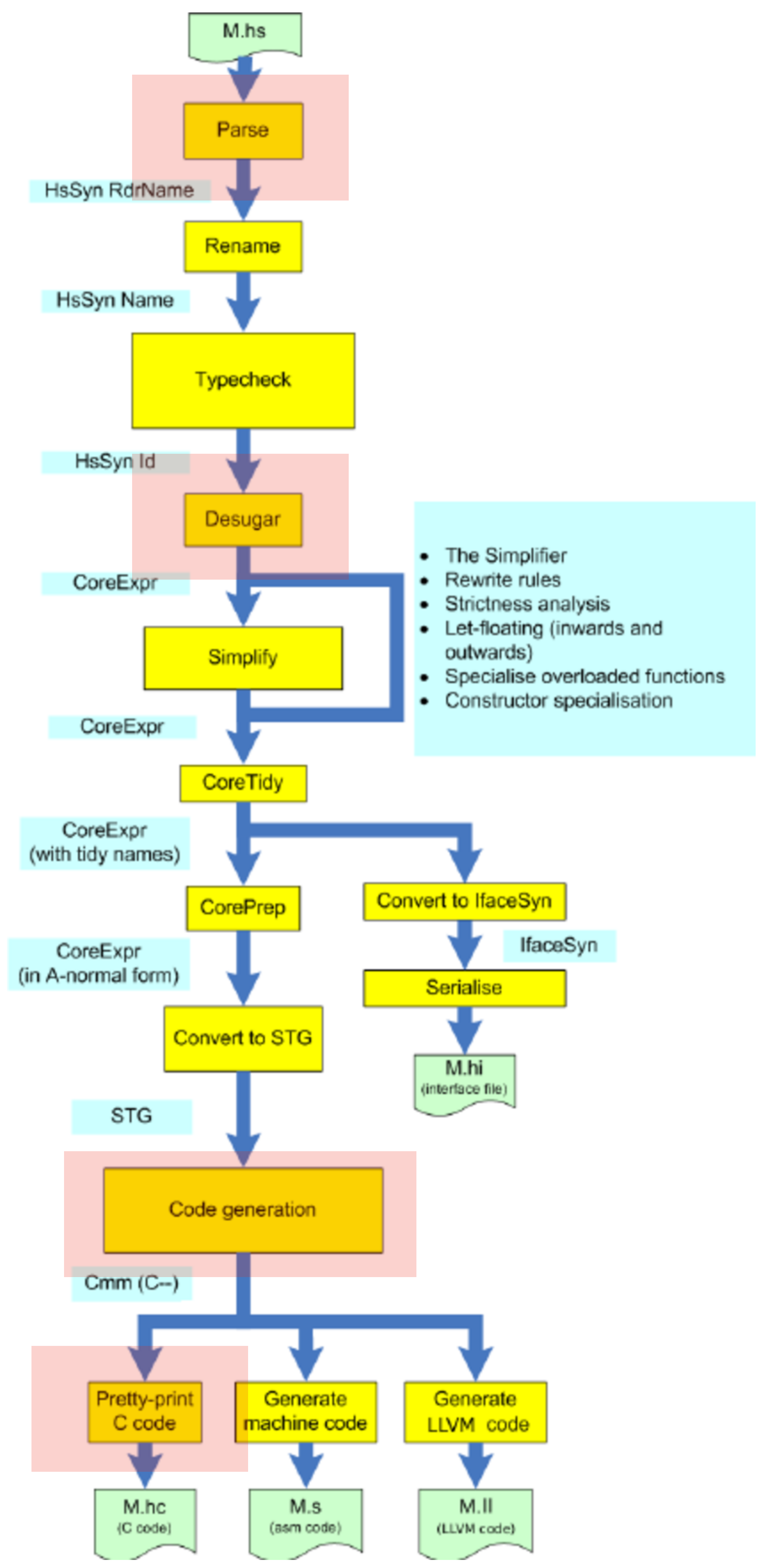


Example Pipelines: GHC (Haskell)



- The Simplifier
- Rewrite rules
- Strictness analysis
- Let-floating (inwards and outwards)
- Specialise overloaded functions
- Constructor specialisation

Example Pipelines: GHC (Haskell)



Example Pipelines: CompCert (C)

Coq
mini-ML
Other languages?
Parser, typechecker, simplifier (CIL)
Type reconstruction
Graph coloring
Printing to asm syntax
Programmed in Caml
C source
Clight
C#minor
Cminor
RTL
LTL
Linear
Mach
PPC
PowerPC assembly
Initial translation
Stack pre-allocation
CFG construction; instruction recognition
Validation
Validation
Linearization of the CFG
Layout of the activation record
Generation of Power PC instructions
Programmed and proved in Coq
Program prover
Model checker
Static analyzer
Dataflow analyses
Constant propagation
Common subexpressions
Register allocation by graph coloring
Data structures (Maps, Sets)
Machine arithmetic
Memory model

Example Pipelines: CompCert (C)

Coq
Other languages?
mini-ML
Parser, typechecker, simplifier (CIL)
Type reconstruction
Graph coloring
Printing to asm syntax
Programmed in Caml
C source
Clight
C#minor
Cminor
RTL
LTL
Linear
Mach
PPC
PowerPC assembly
Initial translation
Stack pre-allocation
CFG construction; instruction recognition
Validation
Validation
Linearization of the CFG
Layout of the activation record
Generation of Power PC instructions
Programmed and proved in Coq
Program prover
Model checker
Static analyzer
Dataflow analyses
Constant propagation
Common subexpressions
Register allocation by graph coloring
Data structures (Maps, Sets)
Machine arithmetic
Memory model

A Remark

This is not a strong distinction

Many languages do a **combination** of interpretation and compilation (we're doing this in project 3)

Things like **just-in-time compilation** make this distinction more complicated

demo

(c to asm)

Project 3 Overview

Source Language

```
 <prog>  ::=  {let <ident> <args> = <expr>}
 <expr>  ::=  () | <num> | <bool> | <ident>
          |   <uop> <expr> | <expr> <bop> <expr>
          |   fun <args1> -> <expr> | <expr> <expr>
          |   let <ident> <args> = <expr> in <expr>
          |   if <expr> then <expr> else <expr>
          |   trace <expr> | ( <expr> )
 <args>  ::=  {<ident>}
<args1>  ::=   <ident> {<ident>}
  <uop>  ::=  not | -
  <bop>  ::=  + | - | * | / | && | || | < | <= | > | >= | = | <>
```

This is a subset of OCaml syntax with an additional **trace** operator.

Target Language

```
<prog>  ::=  {<com>}
<com>   ::=  push <const> | swap | trace
          |  add | sub | mul | div | lt
          |  if <prog> else <prog> end
          |  fun <ident> begin <prog> end | call | return
          |  assign <ident> | lookup <ident>
<const> ::=  <bool> | <nat> | unit
```

This is a subset of our stack-oriented language from Project 2 with less user-friendly syntax.

We also assume a different operational semantics, lexical scoping and immutable variables implemented via continuation passing.

Interlude: Parameter Passing

Recall: Parameter Passing

$$\frac{}{\langle\, [\, F \,,\, C \,,\, Q \,] :: S \,,\, E \,,\, T \,,\, \texttt{call}\ P \,\rangle \longrightarrow \langle\, [\, \mathsf{CC} \,,\, E \,,\, P \,] \,,\, \mathsf{update}(C, F, [\, F \,,\, C \,,\, Q \,]) \,,\, T \,,\, Q \,\rangle}\ \text{(call)}$$

$$\frac{}{\langle\, [\, F \,,\, E \,,\, P \,] :: S \,,\, C \,,\, T \,,\, \texttt{return}\ Q \,\rangle \longrightarrow \langle\, S \,,\, E \,,\, T \,,\, P \,\rangle}\ \text{(return)}$$

Recall: Parameter Passing

$$\frac{}{\langle\, [\, F \,,\, C \,,\, Q \,] :: S \,,\, E \,,\, T \,,\, \texttt{call}\ P \,\rangle \longrightarrow \langle\, [\, \mathsf{CC} \,,\, E \,,\, P \,] \,,\, \mathsf{update}(C, F, [\, F \,,\, C \,,\, Q \,]) \,,\, T \,,\, Q \,\rangle}\ (\mathsf{call})$$

$$\frac{}{\langle\, [\, F \,,\, E \,,\, P \,] :: S \,,\, C \,,\, T \,,\, \texttt{return}\ Q \,\rangle \longrightarrow \langle\, S \,,\, E \,,\, T \,,\, P \,\rangle}\ (\mathsf{return})$$

Our operational semantics has no implicit notion of a function argument.

Recall: Parameter Passing

$$\frac{}{\langle\, [\, F\,,\, C\,,\, Q\,] :: S\,,\, E\,,\, T\,,\, \texttt{call}\ P\,\rangle \longrightarrow \langle\, [\, \mathsf{CC}\,,\, E\,,\, P\,]\,,\, \mathsf{update}(C, F, [\, F\,,\, C\,,\, Q\,])\,,\, T\,,\, Q\,\rangle}\ (\mathsf{call})$$

$$\frac{}{\langle\, [\, F\,,\, E\,,\, P\,] :: S\,,\, C\,,\, T\,,\, \texttt{return}\ Q\,\rangle \longrightarrow \langle\, S\,,\, E\,,\, T\,,\, P\,\rangle}\ (\mathsf{return})$$

Our operational semantics has no implicit notion of a function argument.

Question. *What do we do about that?*

Recall: Parameter Passing

$$\frac{}{\langle\, [\, F \,,\, C \,,\, Q \,] :: S \,,\, E \,,\, T \,,\, \texttt{call}\ P \,\rangle \longrightarrow \langle\, [\, \mathsf{CC} \,,\, E \,,\, P \,] \,,\, \mathsf{update}(C, F, [\, F \,,\, C \,,\, Q \,]) \,,\, T \,,\, Q \,\rangle}\ (\mathsf{call})$$

$$\frac{}{\langle\, [\, F \,,\, E \,,\, P \,] :: S \,,\, C \,,\, T \,,\, \texttt{return}\ Q \,\rangle \longrightarrow \langle\, S \,,\, E \,,\, T \,,\, P \,\rangle}\ (\mathsf{return})$$

Our operational semantics has no implicit notion of a function argument.

Question. *What do we do about that?*

We can put arguments and return values <u>under</u> the current continuation.

Example: Parameter Passing

Stack:

Program:

```
fun SQUARE begin
  swap assign X
  lookup X lookup X
  mul
  swap return
end
push 2 swap call
assign Y
```

Env:

Example: Parameter Passing

Stack:

```
closure {
  name: SQUARE
  captured: []
  prog:
  swap assign X lookup X
  lookup X mul swap return
}
```

Program:

```
push 2 swap call
assign Y
```

Env:

Example: Parameter Passing

Stack:

```
2
```

```
closure {
  name: SQUARE
  captured: []
  prog:
  swap assign X lookup X
  lookup X mul swap return
}
```

Program:

```
swap call
assign Y
```

Env:

Example: Parameter Passing

Stack:

```
closure {
  name: SQUARE
  captured: []
  prog:
  swap assign X lookup X
  lookup X mul swap return
}
```

```
2
```

Program:

```
call
assign Y
```

Env:

Example: Parameter Passing

Stack:

```
closure {
  name: CC
  captured: []
  prog:
  assign Y
}
```

```
2
```

Program:

```
swap assign X
lookup X lookup X
mul
swap return
```

Env:

Example: Parameter Passing

Stack:

```
2
```

```
closure {
  name: CC
  captured: []
  prog:
  assign Y
}
```

Program:

```
assign X
lookup X lookup X
mul
swap return
```

Env:

Example: Parameter Passing

Stack:

```
closure {
  name: CC
  captured: []
  prog:
  assign Y
}
```

Program:

```
lookup X lookup X
mul
swap return
```

Env: X $\mapsto$ 2

Example: Parameter Passing

Stack:

```
2
```

```
closure {
  name: CC
  captured: []
  prog:
  assign Y
}
```

Program:

```
lookup X
mul
swap return
```

Env: X $\mapsto$ 2

Example: Parameter Passing

Stack:

```
2
```

```
2
```

```
closure {
  name: CC
  captured: []
  prog:
  assign Y
}
```

Program:

```
mul
swap return
```

Env: $X \mapsto 2$

Example: Parameter Passing

Stack:

$\boxed{4}$

```
closure {
  name: CC
  captured: []
  prog:
  assign Y
}
```

Program:

`swap return`

Env: X ↦ 2

Example: Parameter Passing

Stack:

```
closure {
  name: CC
  captured: []
  prog:
  assign Y
}
```

```
4
```

Program:

```
return
```

Env: X ↦ 2

Example: Parameter Passing

Stack:

$\boxed{4}$

Program:

```
assign Y
```

Env:

Example: Parameter Passing

Stack:

Program:

Env: Y $\mapsto$ 4

back to compilation...

Simple Example

```
let k x y = x
let _ = trace (k 5 10)
```

Simple Example

```
(fun k ->
  (fun _ -> ())
    (trace (k 5 10)))
  (fun x -> fun y -> x)
```

(desugared)

Simple Example

what's the deal with these identifiers?

```
fun C begin swap assign AX fun C begin swap assign AY
lookup AX swap return end
swap return end
fun C begin swap assign AK
push 10 push 5 lookup AK call call trace push unit
fun C begin swap assign BK
push unit swap return
end
call swap return
end call
```

(translated)

Simple Example

```
let k x y = x
let _ = trace (k 5 10)
```

$\longrightarrow$

```
fun C begin swap assign AX
  fun C begin swap assign AY
    lookup AX
    swap return
  end
  swap return
end
fun C begin swap assign AK
  push 10
  push 5
  lookup AK call
  call trace push unit
  fun C begin swap assign BK
    push unit
    swap return
  end
  call swap return
end call
```

How do we get here?

Simple Example

```
let k x y = x
let _ = trace (k 5 10)
```

$\longrightarrow$

```
fun C begin swap assign AX
  fun C begin swap assign AY
    lookup AX
    swap return
  end
  swap return
end
fun C begin swap assign AK
  push 10
  push 5
  lookup AK call
  call trace push unit
  fun C begin swap assign BK
    push unit
    swap return
  end
  call swap return
end call
```

How do we get here?

Simple Example

```
let k x y = x
let _ = trace (k 5 10)
```

$\longrightarrow$

```
fun C begin swap assign AX
  fun C begin swap assign AY
    lookup AX
    swap return
  end
  swap return
end
fun C begin swap assign AK
  push 10
  push 5
  lookup AK call
  call trace push unit
  fun C begin swap assign BK
    push unit
    swap return
  end
  call swap return
end call
```

How do we get here?

Simple Example

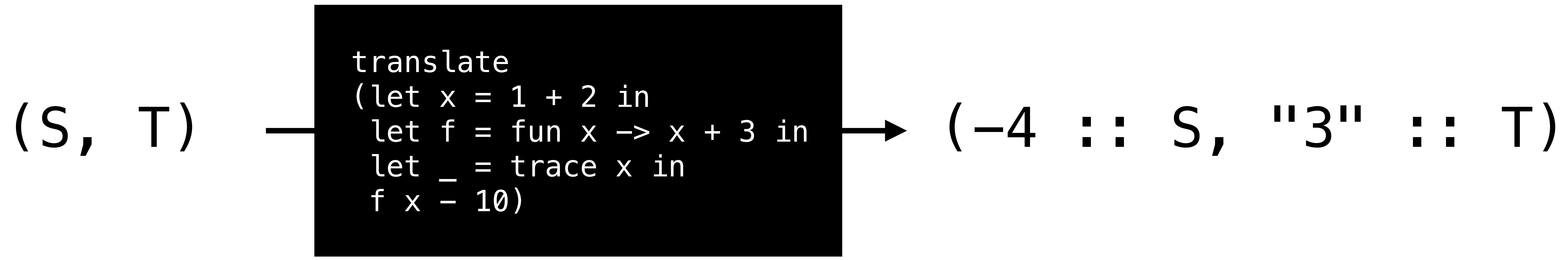
```
let k x y = x
let _ = trace (k 5 10)
```

→

```
fun C begin swap assign AX
  fun C begin swap assign AY
    lookup AX
    swap return
  end
  swap return
end
fun C begin swap assign AK
  push 10
  push 5
  lookup AK call
  call trace push unit
  fun C begin swap assign BK
    push unit
    swap return
  end
  call swap return
end call
```

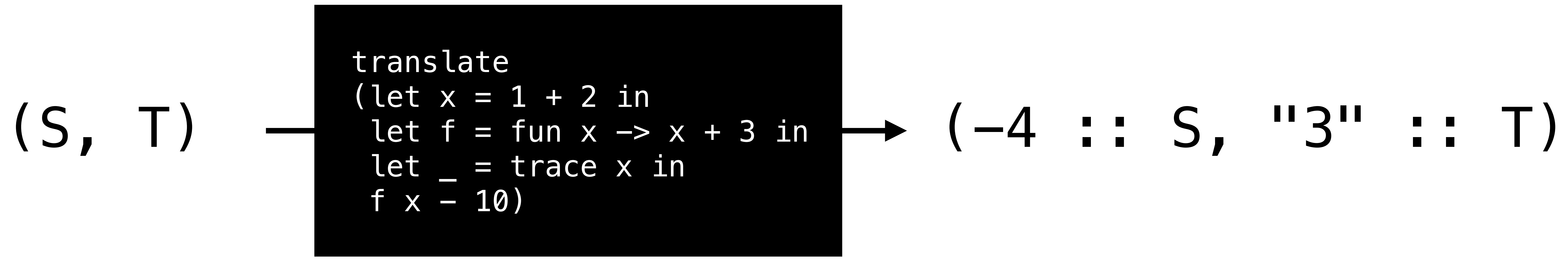
How do we get here?

High Level



```
translate
(let x = 1 + 2 in
 let f = fun x -> x + 3 in
 let _ = trace x in
 f x − 10)
```

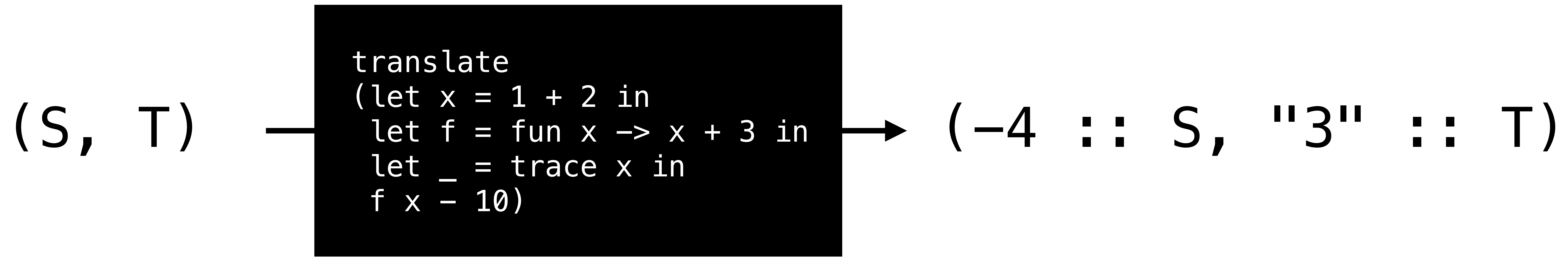
High Level



```
translate
(let x = 1 + 2 in
 let f = fun x -> x + 3 in
 let _ = trace x in
 f x - 10)
```

desugaring is a just matter of replacing syntax

High Level



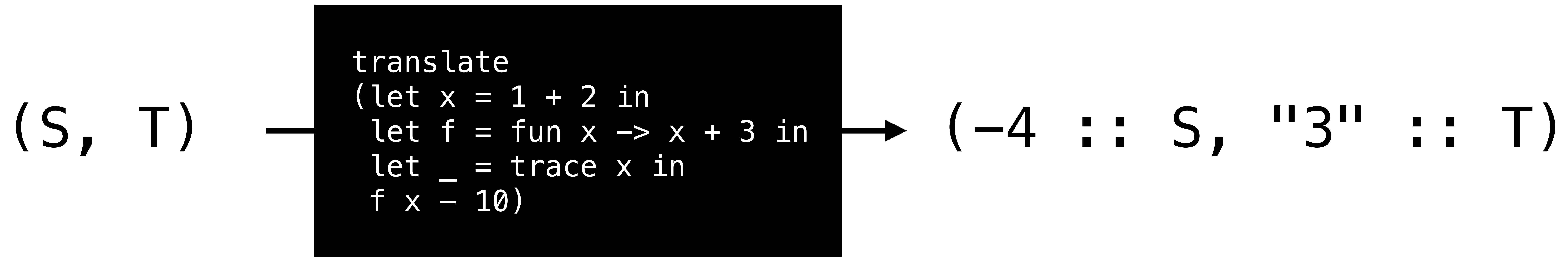
```
translate
(let x = 1 + 2 in
 let f = fun x -> x + 3 in
 let _ = trace x in
 f x − 10)
```

`**desugaring** is a just matter of replacing syntax`

desugaring is a just matter of replacing syntax

serializing is a just matter of representing syntax as a string

High Level



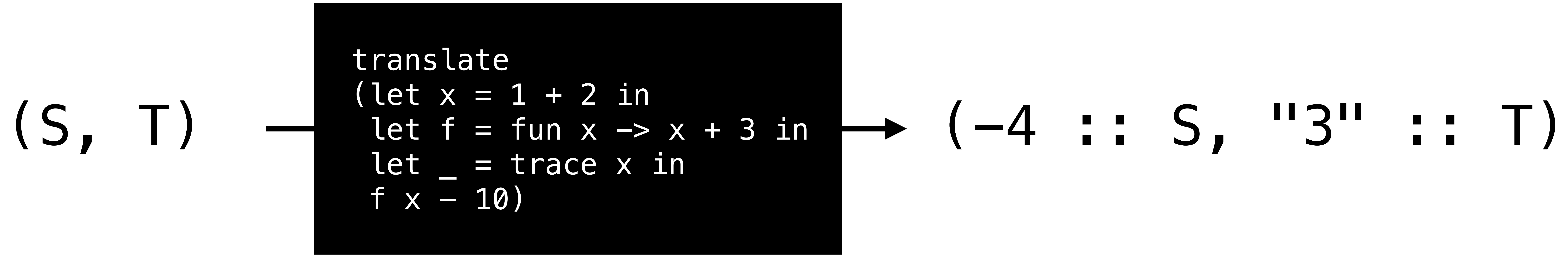
```
translate
(let x = 1 + 2 in
 let f = fun x -> x + 3 in
 let _ = trace x in
 f x - 10)
```

desugaring is a just matter of replacing syntax

serializing is a just matter of representing syntax as a string

translating is is the tricky part:

High Level



```
translate
(let x = 1 + 2 in
 let f = fun x -> x + 3 in
 let _ = trace x in
 f x - 10)
```

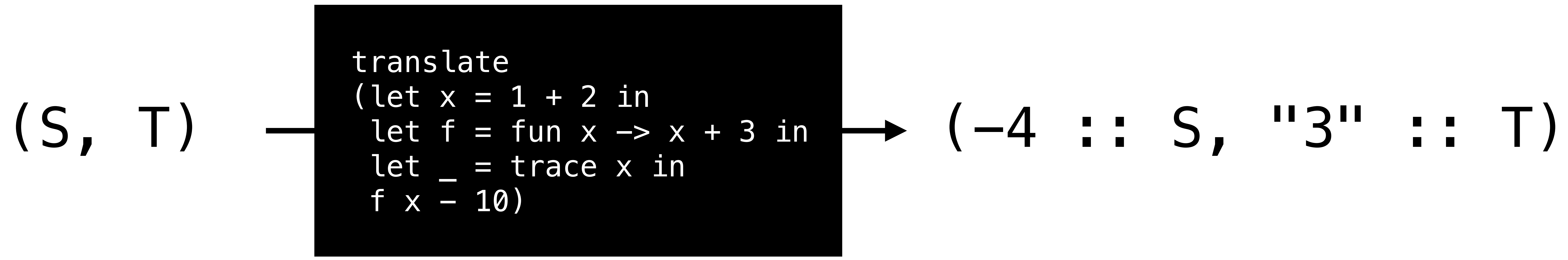
desugaring is a just matter of replacing syntax

serializing is a just matter of representing syntax as a string

translating is is the tricky part:

think of an expression as being translated to a black-box program which leaves just the value of the expression on the stack

High Level



```
translate
(let x = 1 + 2 in
 let f = fun x -> x + 3 in
 let _ = trace x in
 f x - 10)
```

desugaring is a just matter of replacing syntax

serializing is a just matter of representing syntax as a string

translating is is the tricky part:

think of an expression as being translated to a black-box program which leaves just the value of the expression on the stack

(The program may also need to update the trace)

Simple Example (in more detail)

```
let k x y = x
let _ = trace (k 5 10)
```

Simple Example (in more detail)

```
let k x y = x in
let _ = trace (k 5 10) in
()
```

Desugaring should replace a sequence of let-definitions into a sequence of let-bindings for an for a unit

Simple Example (in more detail)

```
let k = fun x -> fun y -> x in
let _ = trace (k 5 10) in
()
```

Desugaring replace let-binding arguments with anonymous functions

Simple Example (in more detail)

```
(fun k ->
  let _ = trace (k 5 10) in
  ())
  (fun x -> fun y -> x)
```

```
Desugaring replace let-binding with function applications
```

Simple Example (in more detail)

```
(fun k ->
  (fun _ ->
    ())
    (trace (k 5 10)))
  (fun x -> fun y -> x)
```

Desugaring replace let-binding with **function applications**

demo

(let's take a moment to understand this)

Simple Example (in more detail)

```
(fun k ->
  (fun _ ->
    ())
    (trace (k 5 10)))
  (fun x -> fun y -> x)
```

Desugaring replace let-binding with **function applications**

Simple Example (in more detail)

```
(fun k ->
  (fun _ -> ())
    (trace (k 5 10)))
  (fun x -> fun y -> x)
```

(desugared)

Simple Example (in more detail)

```
[translate (fun x -> fun y -> x)]
[translate
  (fun k ->
    (fun _ ->
      ())
    (trace (k 5 10))
]
call
```

```
translating should replace function applications with calls commands in
our stack-oriented language
```

Simple Example (in more detail)

```
push arg. to stack [translate (fun x -> fun y -> x)]
                   [translate
                     (fun k ->
                       (fun _ ->
                         ())
                       (trace (k 5 10))
                   ]
                   call
```

translating should replace function applications with calls commands in our stack-oriented language

Simple Example (in more detail)

```
push arg. to stack [translate (fun x -> fun y -> x)]
                   [translate
                     (fun k ->
                       (fun _ ->
push fun. to stack       ())
                       (trace (k 5 10))
                   ]
                   call
```

translating should replace function applications with calls commands in our stack-oriented language

Simple Example (in more detail)

```
fun C begin swap assign AX
  [translate fun y -> x]
swap return
end
[translate
  (fun k ->
    (fun _ ->
      ())
    (trace (k 5 10))
]
call
```

```
translating should replace function anonymous functions with function
definitions in our stack-oriented language
```

Simple Example (in more detail)

```
                                  link formal and actual parameter
fun C begin swap assign AX
  [translate fun y -> x]
swap return
end
[translate
  (fun k ->
    (fun _ ->
      ())
    (trace (k 5 10))
]
call
```

translating should replace function anonymous functions with function definitions in our stack-oriented language

Simple Example (in more detail)

```
                    link formal and actual parameter
fun C begin swap assign AX
  [translate fun y -> x]
swap return
end  put the return value on the stack
[translate
  (fun k ->
    (fun _ ->
      ())
    (trace (k 5 10))
]
call
```

translating should replace function anonymous functions with function definitions in our stack-oriented language

Simple Example (in more detail)

```
fun C begin swap assign AX
  fun C begin swap assign AY
    [translate x]
    swap return
  end
  swap return
end
[translate
  (fun k ->
    (fun _ ->
      ())
    (trace (k 5 10))
]
call
```

translating should replace function anonymous functions with function definitions in our stack-oriented language

Simple Example (in more detail)

```
fun C begin swap assign AX
  fun C begin swap assign AY
    lookup AX
    swap return
  end
  swap return
end
[translate
  (fun k ->
    (fun _ ->
      ())
    (trace (k 5 10))
]
call
```

translating should replace variables with lookups in the environment

Simple Example (in more detail)

```
fun C begin swap assign AX
  fun C begin swap assign AY
    lookup AX refers to the formal parameter
    swap return
  end
  swap return
end
[translate
  (fun k ->
    (fun _ ->
      ())
    (trace (k 5 10))
]
call
```

translating should replace variables with lookups in the environment

One Last Point: Evaluation Order

$$\frac{(\ T\ ,\ e_2\ ) \longrightarrow (\ T'\ ,\ e_2'\ )}{(\ T\ ,\ e_1 + e_2\ ) \longrightarrow (\ T'\ ,\ e_1 + e_2'\ )}\ \text{(addRight)} \qquad \frac{(\ T\ ,\ e_1\ ) \longrightarrow (\ T'\ ,\ e_1'\ ) \qquad v \in \mathbb{Z}}{(\ T\ ,\ e_1 + v\ ) \longrightarrow (\ T'\ ,\ e_1' + v\ )}\ \text{(addLeft)}$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{(\ T\ ,\ m + n\ ) \longrightarrow (\ T\ ,\ m + n\ )}\ \text{(addNum)}$$

The order in which you evaluate (and, hence, translate) arguments is implicit in the operational semantics.

Question. *In which order should you evaluate the arguments to the '+' operator?*

Understanding Check

```
1 + (2 + 30)
```

What should the above expression be translated to?

And why does it matter?

Answer

```
push 30
push 2
add
push 1
add
```

It matters because the arguments could affect the trace. Consider:

```
(let _ = trace 1 in 1) +
((let _ = trace 2 in 2) + (let _ = trace 30 in 30))
```

What should the trace look like after evaluating this expression?

demo

(more example if there's time)