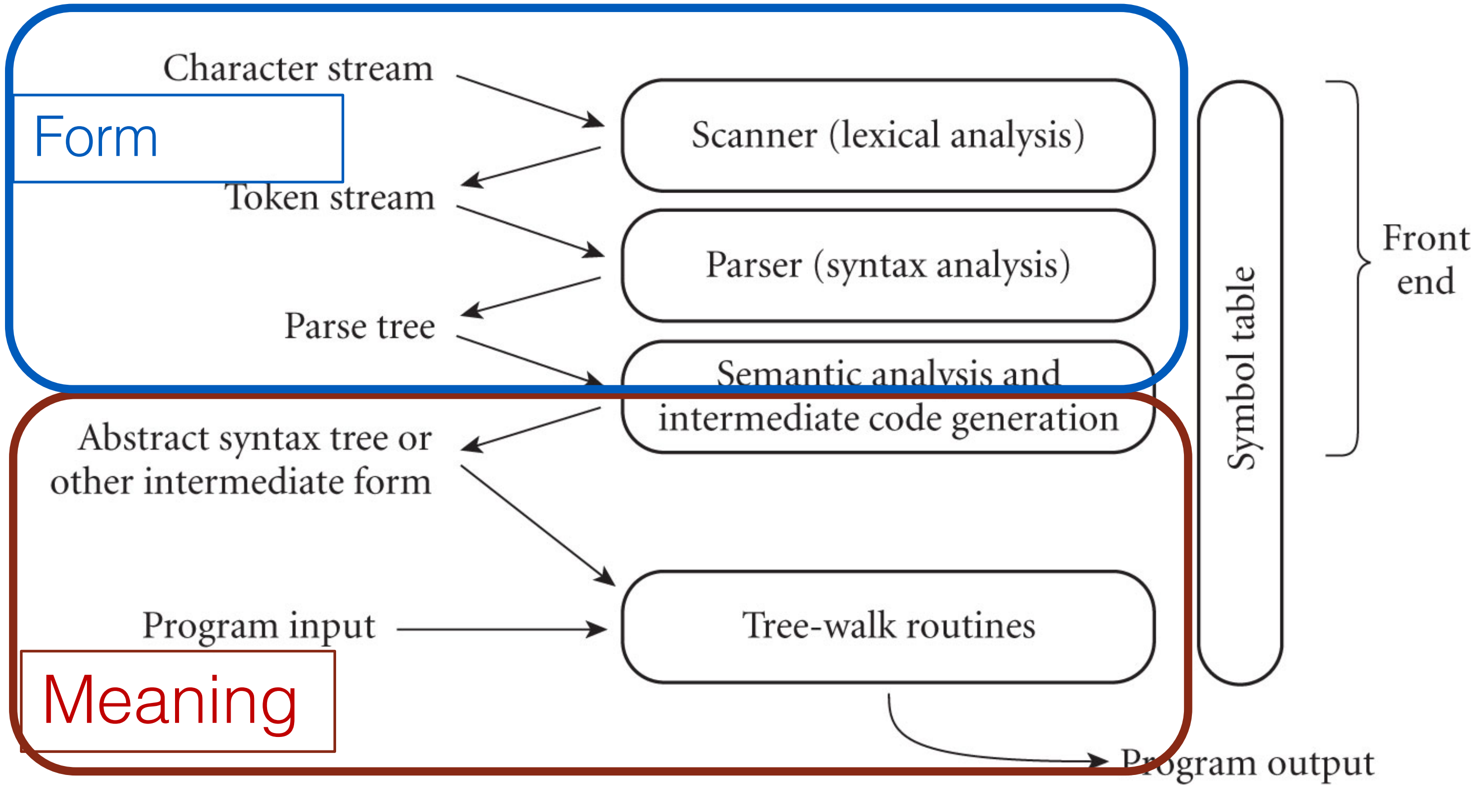
CS 320 : Compiler

Marco Gaboardi
CDS 1019
gaboardi@bu.edu

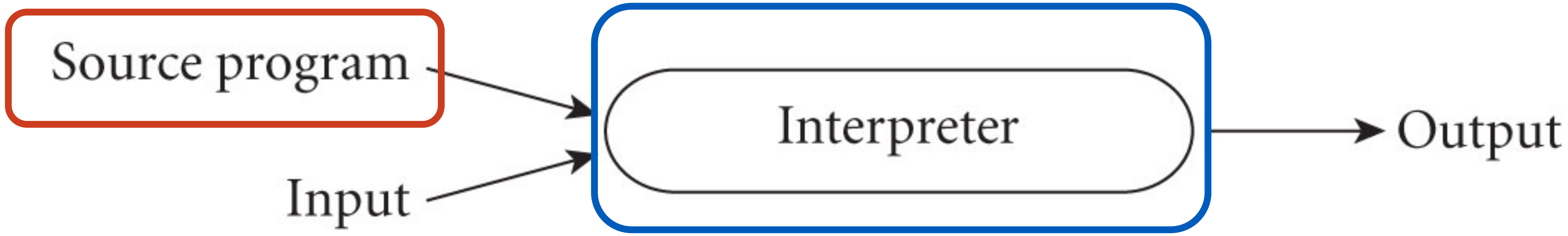
Form and Meaning



Learning Goals for today

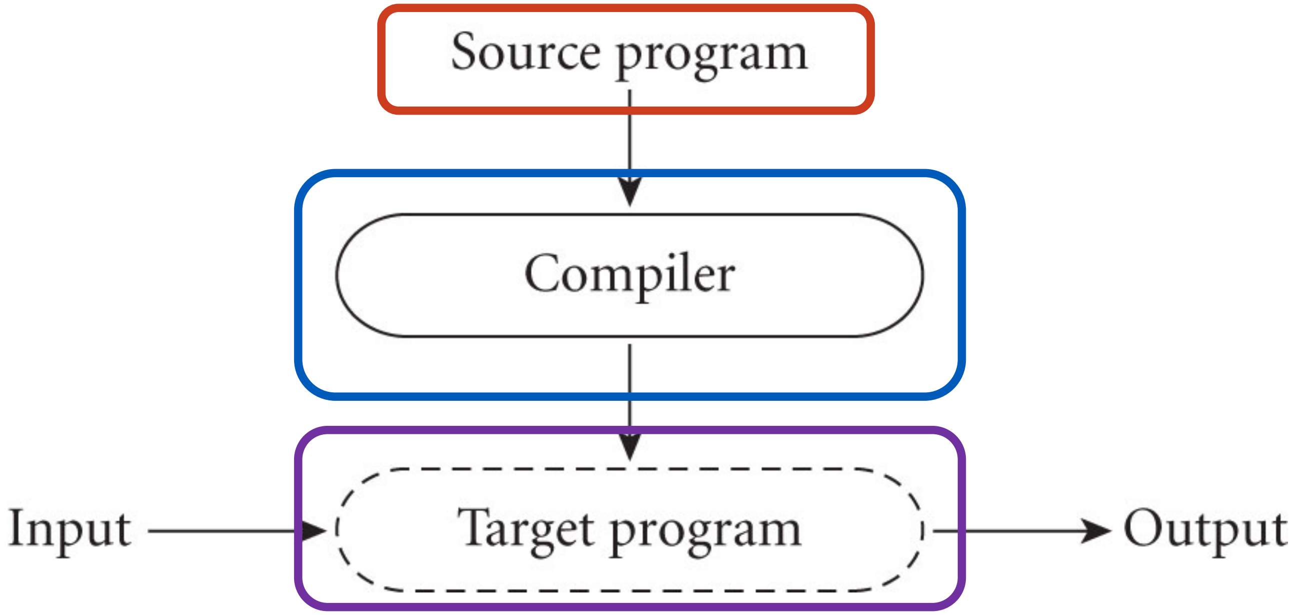
- Understanding the basics of a compiler.
- Understanding the difference with an interpreter.

Pure Interpretation



An interpreter is a program that accepts a source program and its input and runs it immediately to produce the output.

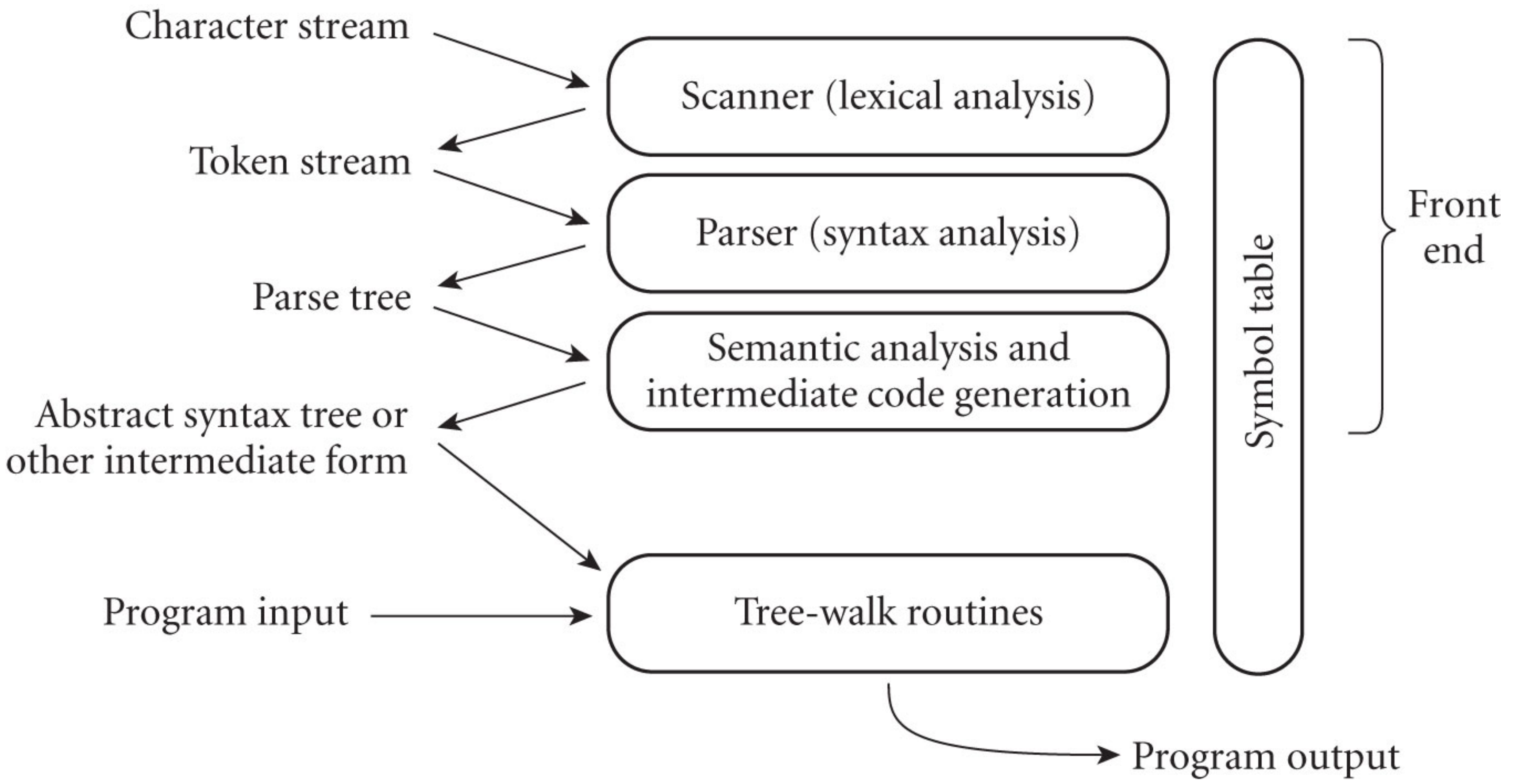
Pure Compilation



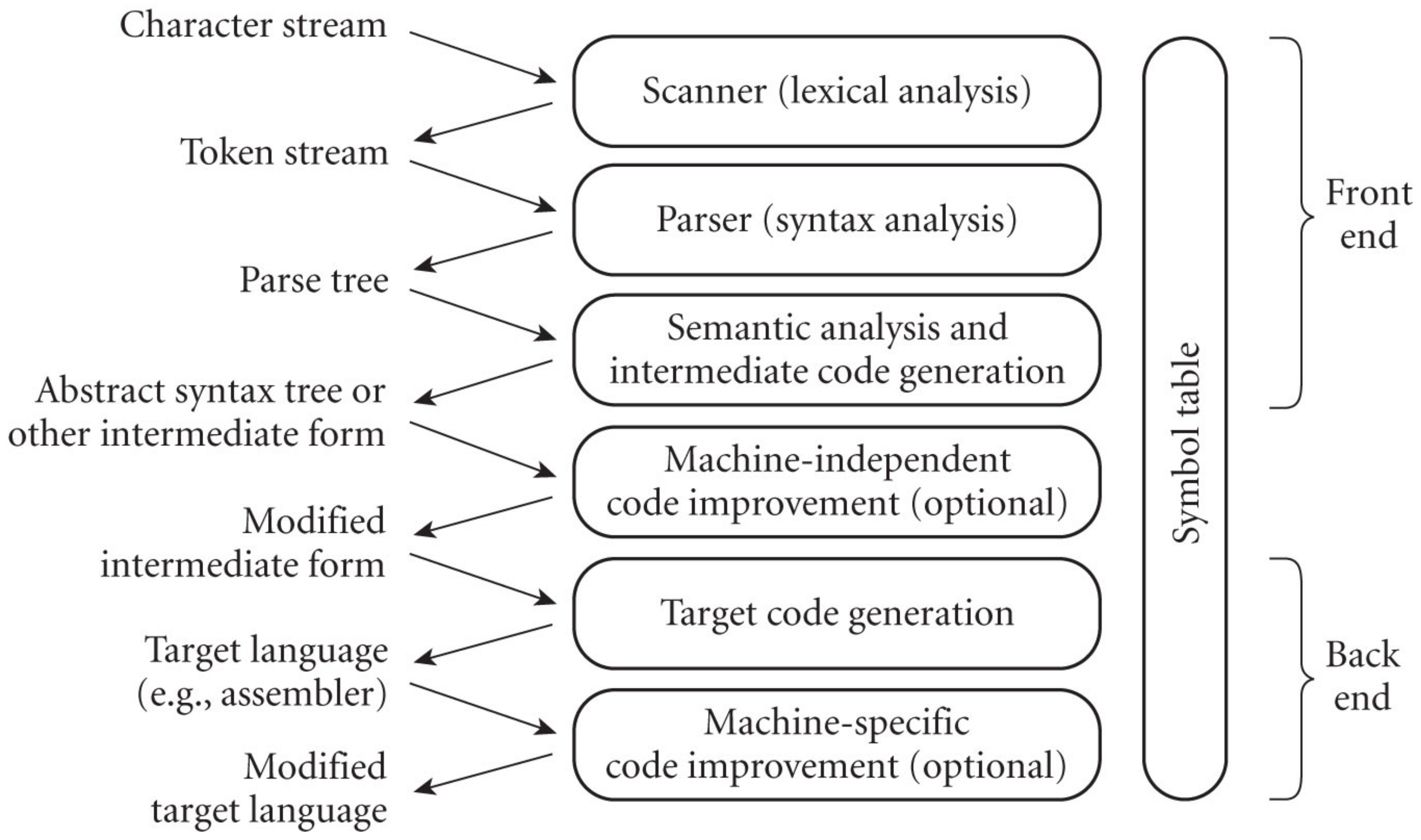
A compiler is a program that translates from a source program from an high-level language into a low-level language.

What are the phases of an interpreter or a compiler?

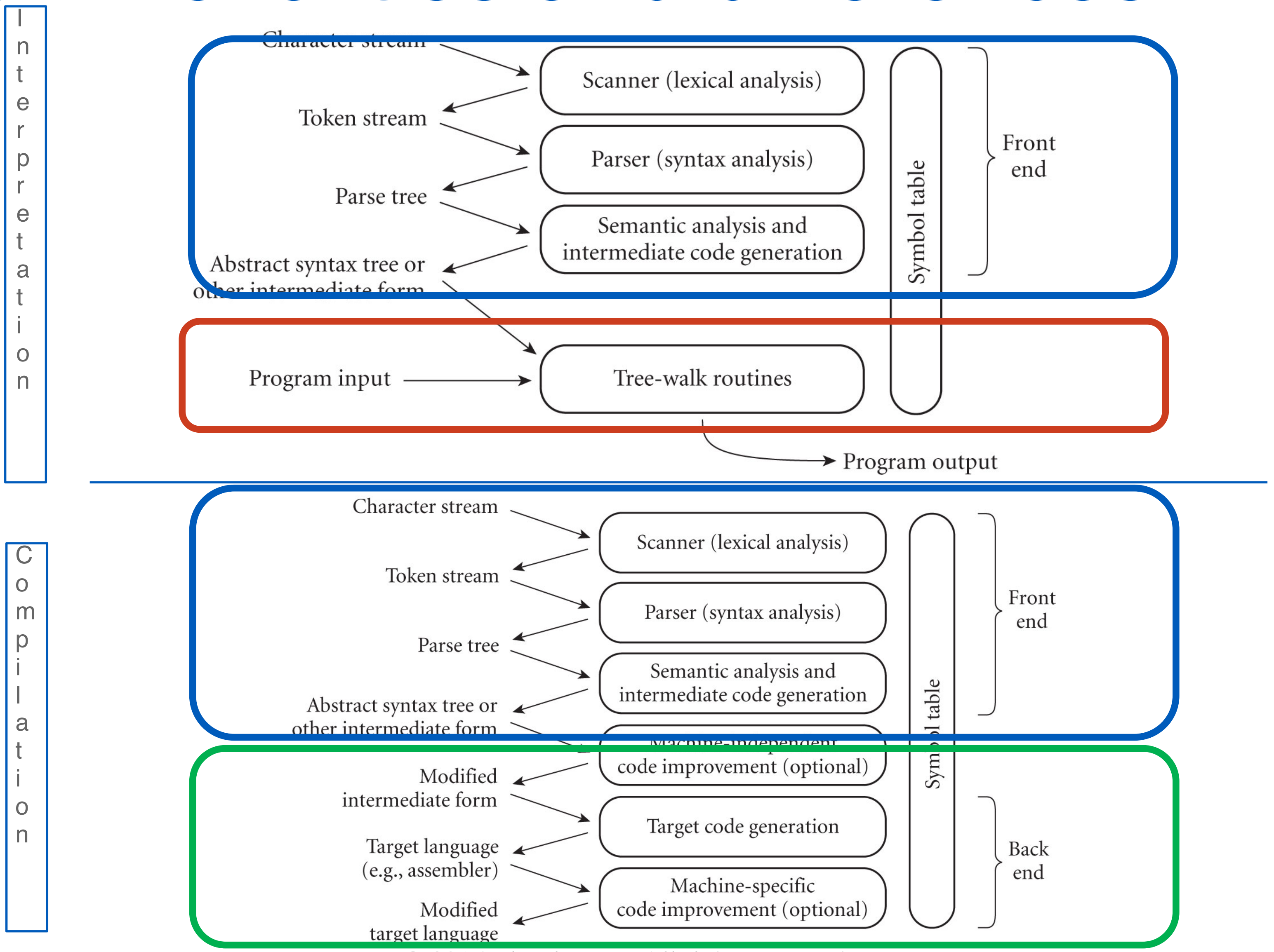
Pure Interpretation



Pure Compilation

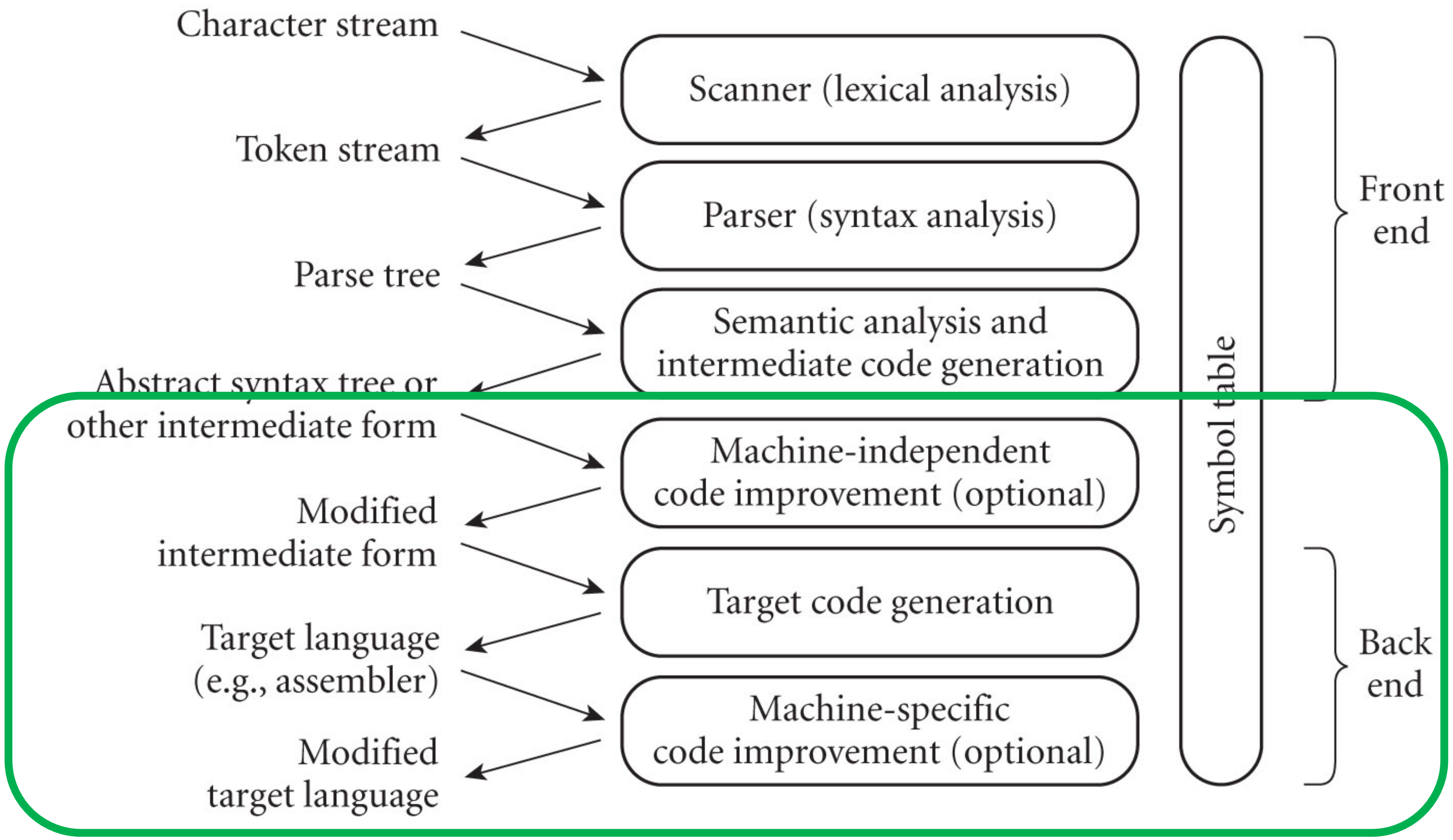

Character stream
Token stream
Parse tree
Abstract syntax tree or other intermediate form
Modified intermediate form
Target language (e.g., assembler)
Modified target language
Scanner (lexical analysis)
Parser (syntax analysis)
Semantic analysis and intermediate code generation
Machine-independent code improvement (optional)
Target code generation
Machine-specific code improvement (optional)
Symbol table
Front end
Back end

Commonalities and differences

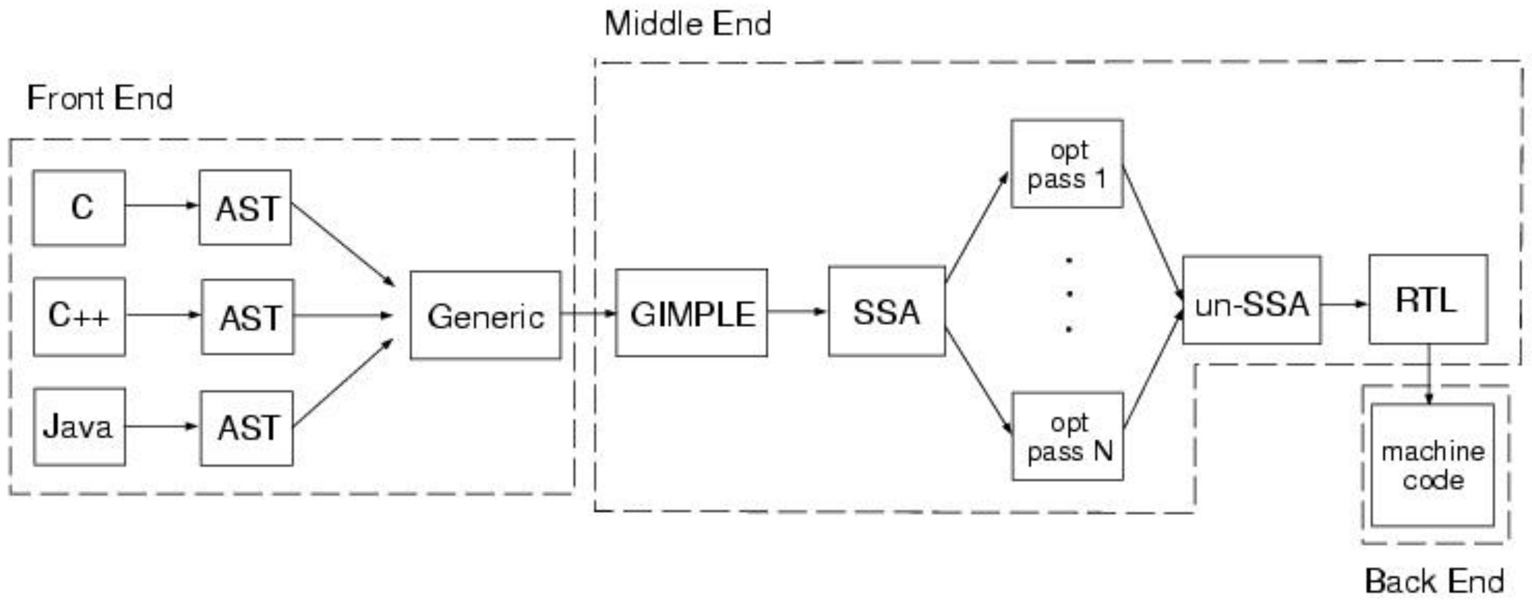


I n t e r p r e t a t i o n
Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis and intermediate code generation
Abstract syntax tree or other intermediate form
Symbol table
Front end
Program input
Tree-walk routines
Program output
C o m p i l a t i o n
Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis and intermediate code generation
Abstract syntax tree or other intermediate form
Machine-independent code improvement (optional)
Modified intermediate form
Target code generation
Target language (e.g., assembler)
Machine-specific code improvement (optional)
Modified target language
Symbol table
Front end
Back end

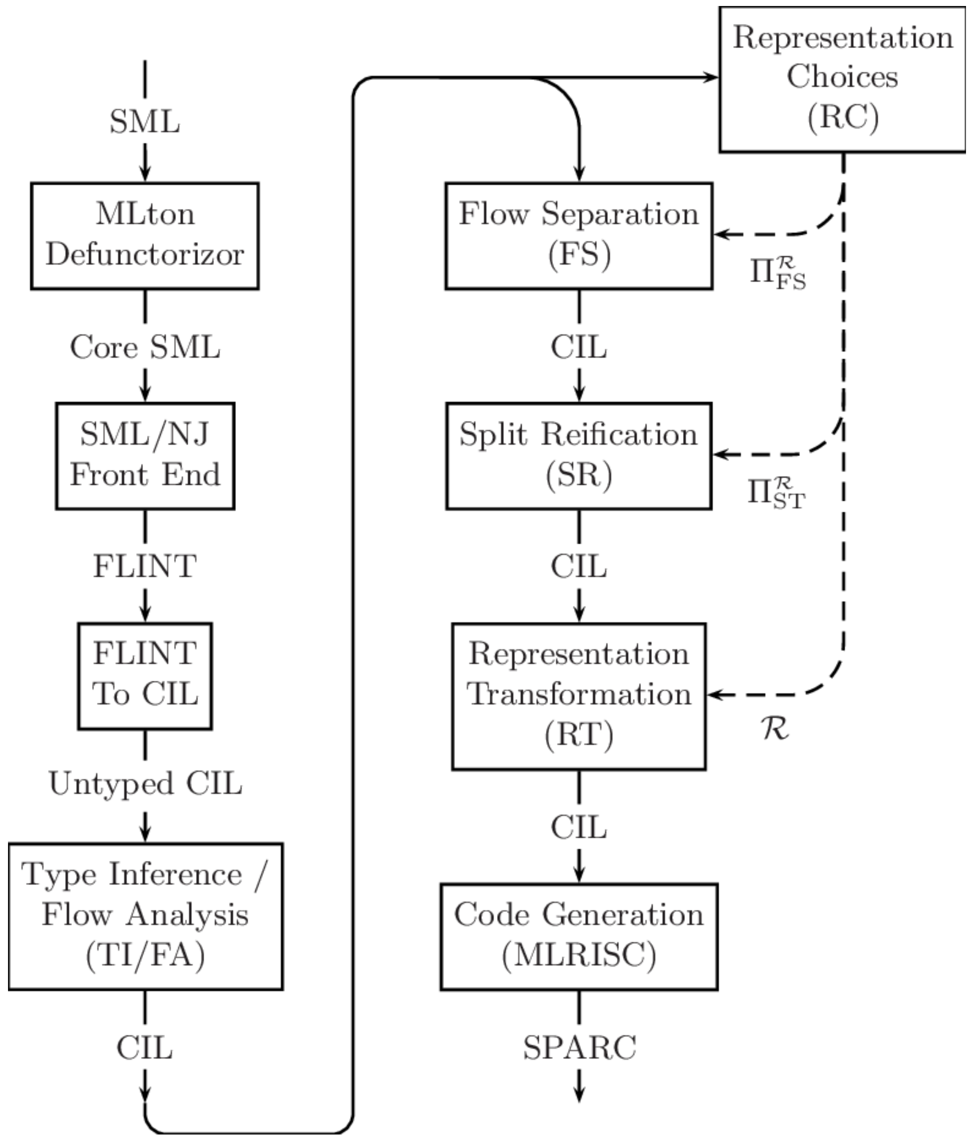
Pure Compilation


Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis and intermediate code generation
Abstract syntax tree or other intermediate form
Machine-independent code improvement (optional)
Modified intermediate form
Target code generation
Target language (e.g., assembler)
Machine-specific code improvement (optional)
Modified target language
Symbol table
Front end
Back end

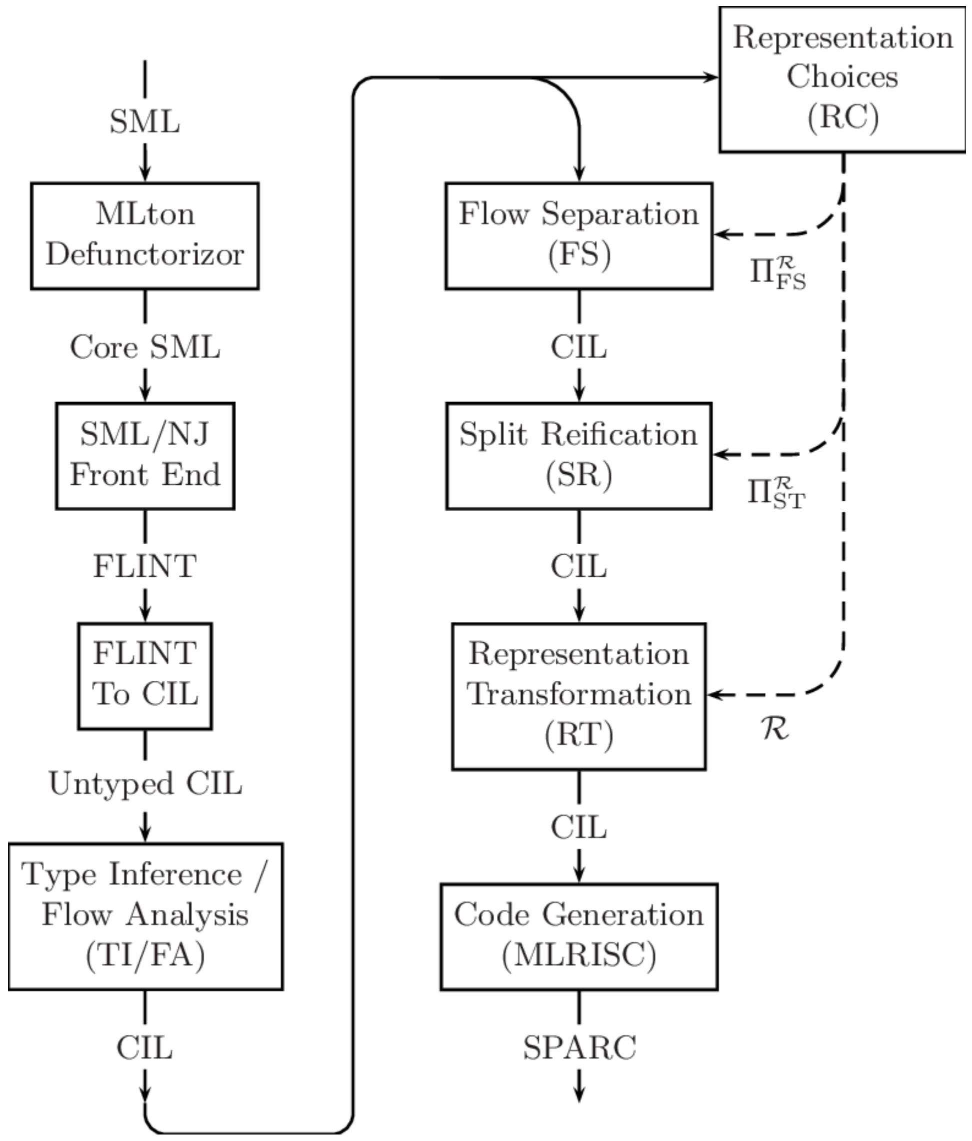
Some examples - gcc



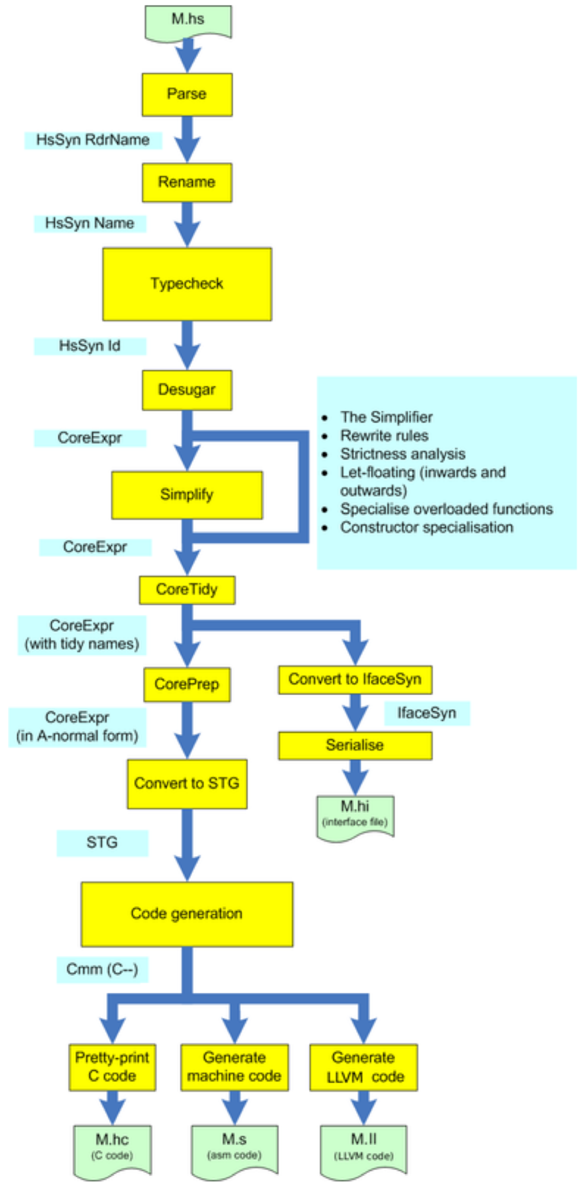
Some examples - GCC



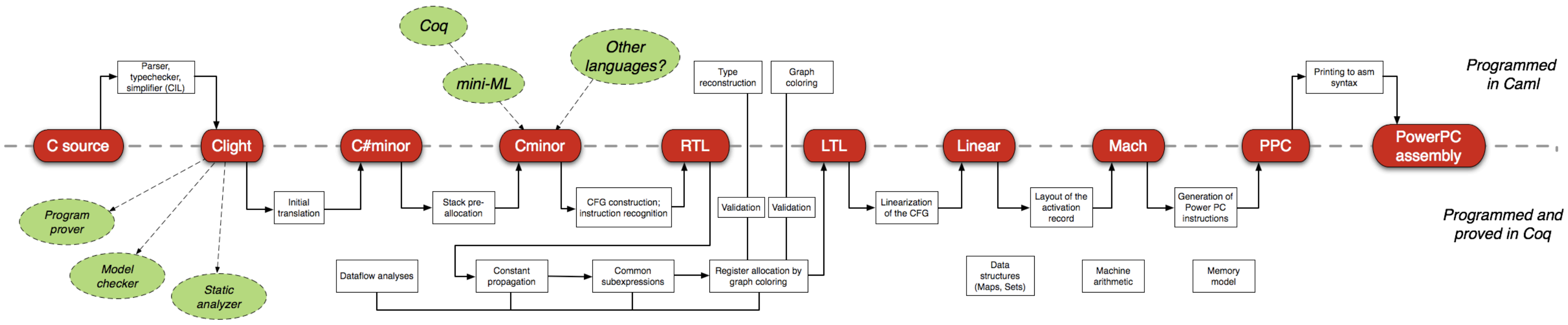
Some examples - SML



Some examples - GHC

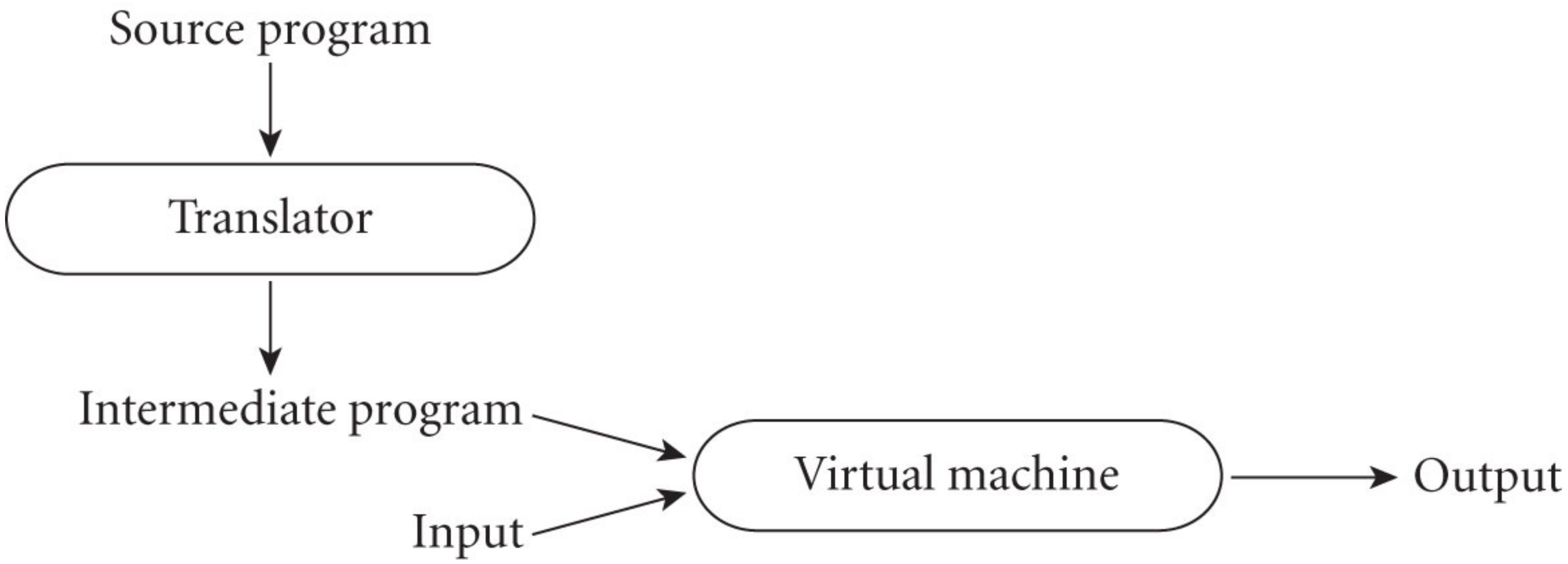


Some examples - GHC



Other approaches: e.g. Mixing Compilation and Interpretation

Mixing



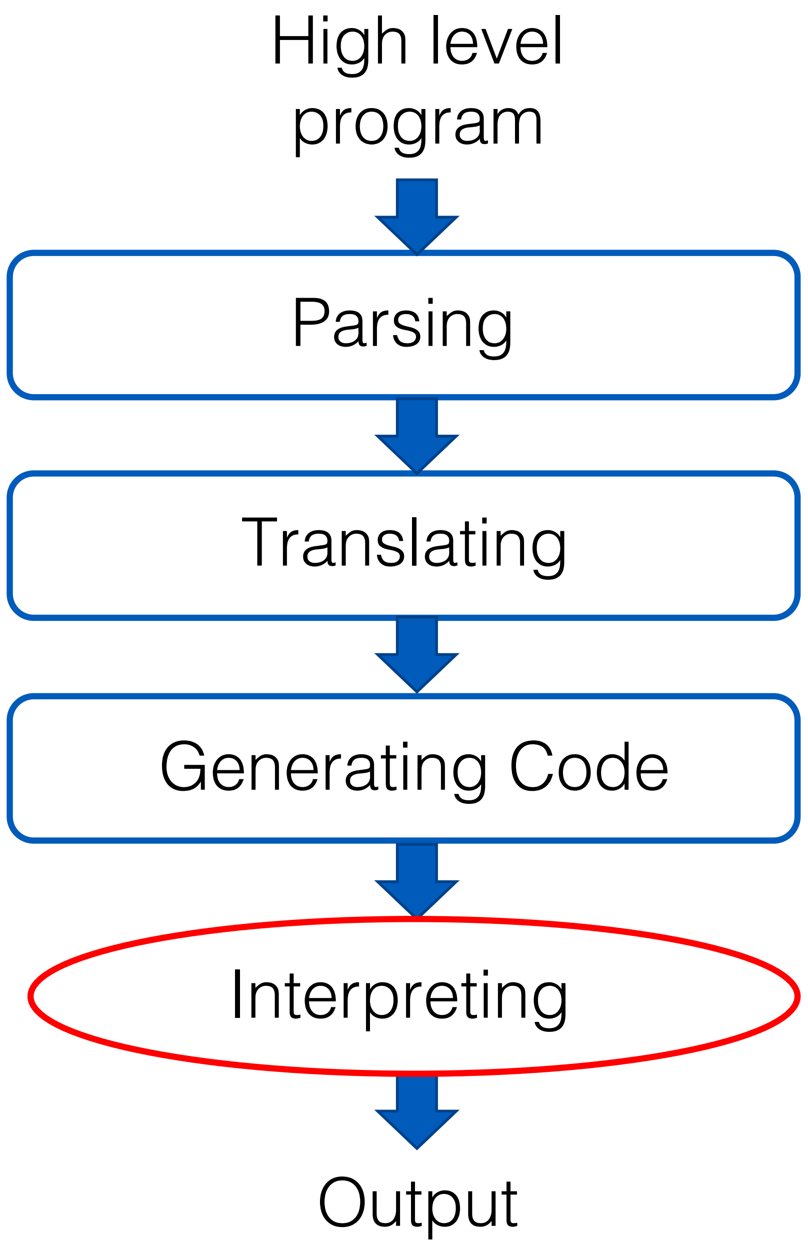
High-level functional language

```
…

<expr> ::= fun <var> <var> -> <expr> | <expr> <expr> |
            <expr> + <expr> | <int>
            let <var> = <expr> in <expr>

            …
```

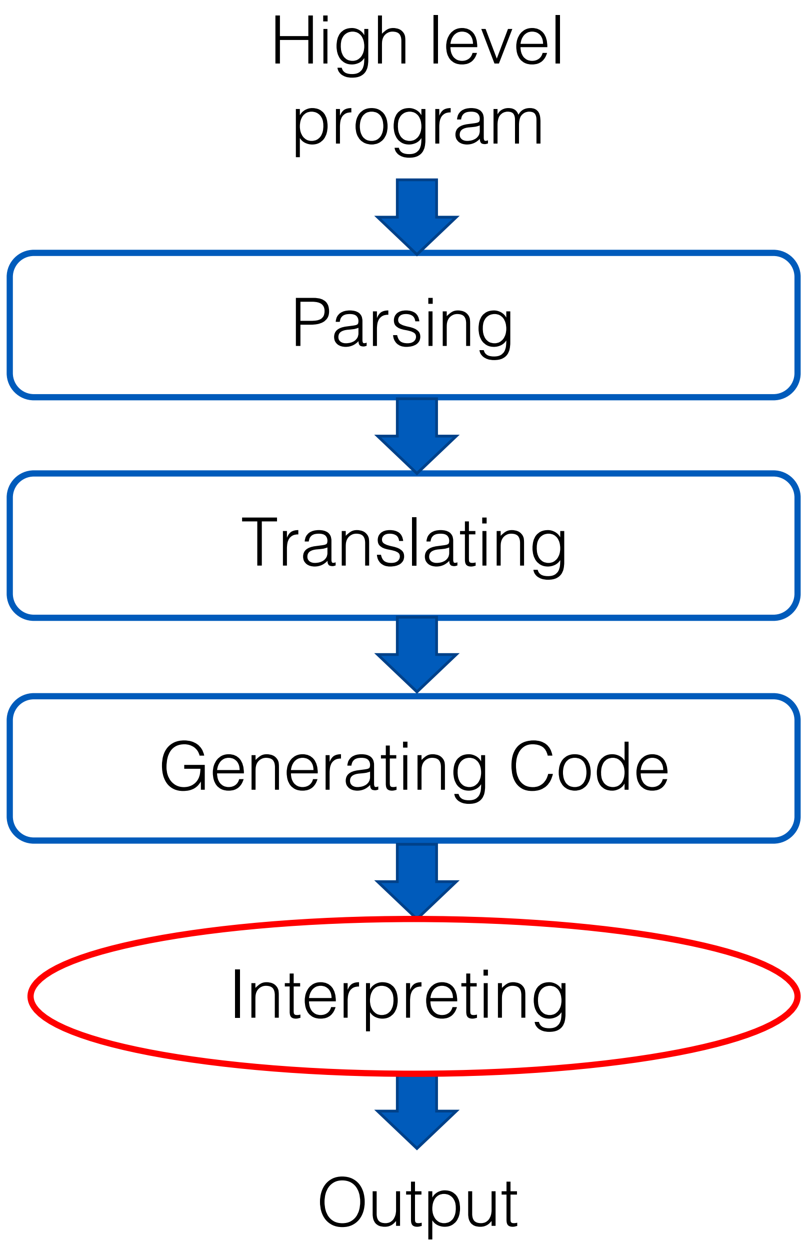
Stack-based language

```
…
<com>    ::= Push <const> | Add |
             | Bind | Lookup
             | Fun <coms> End  | Call | Return
```

Our compiler overview

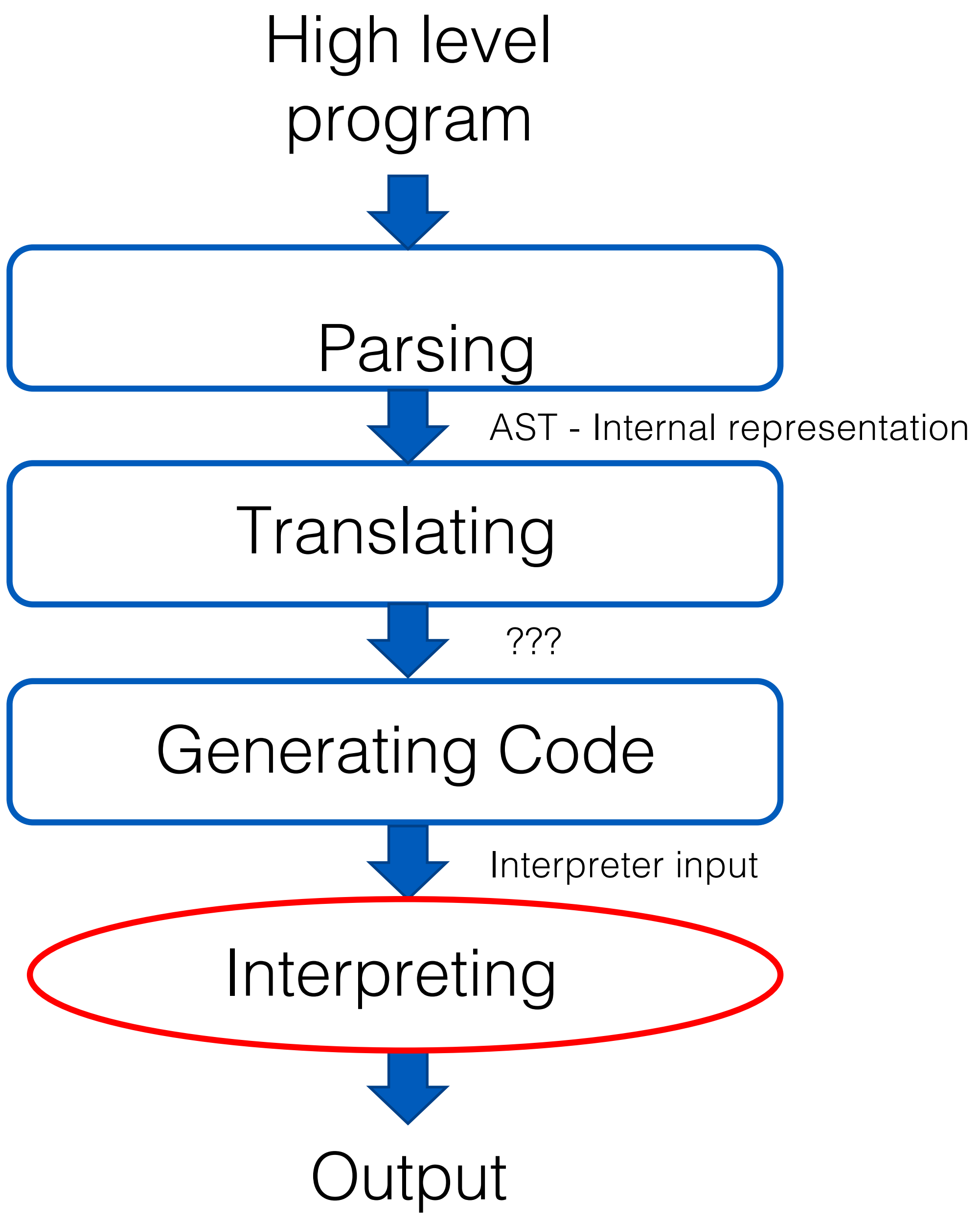


Our compiler overview

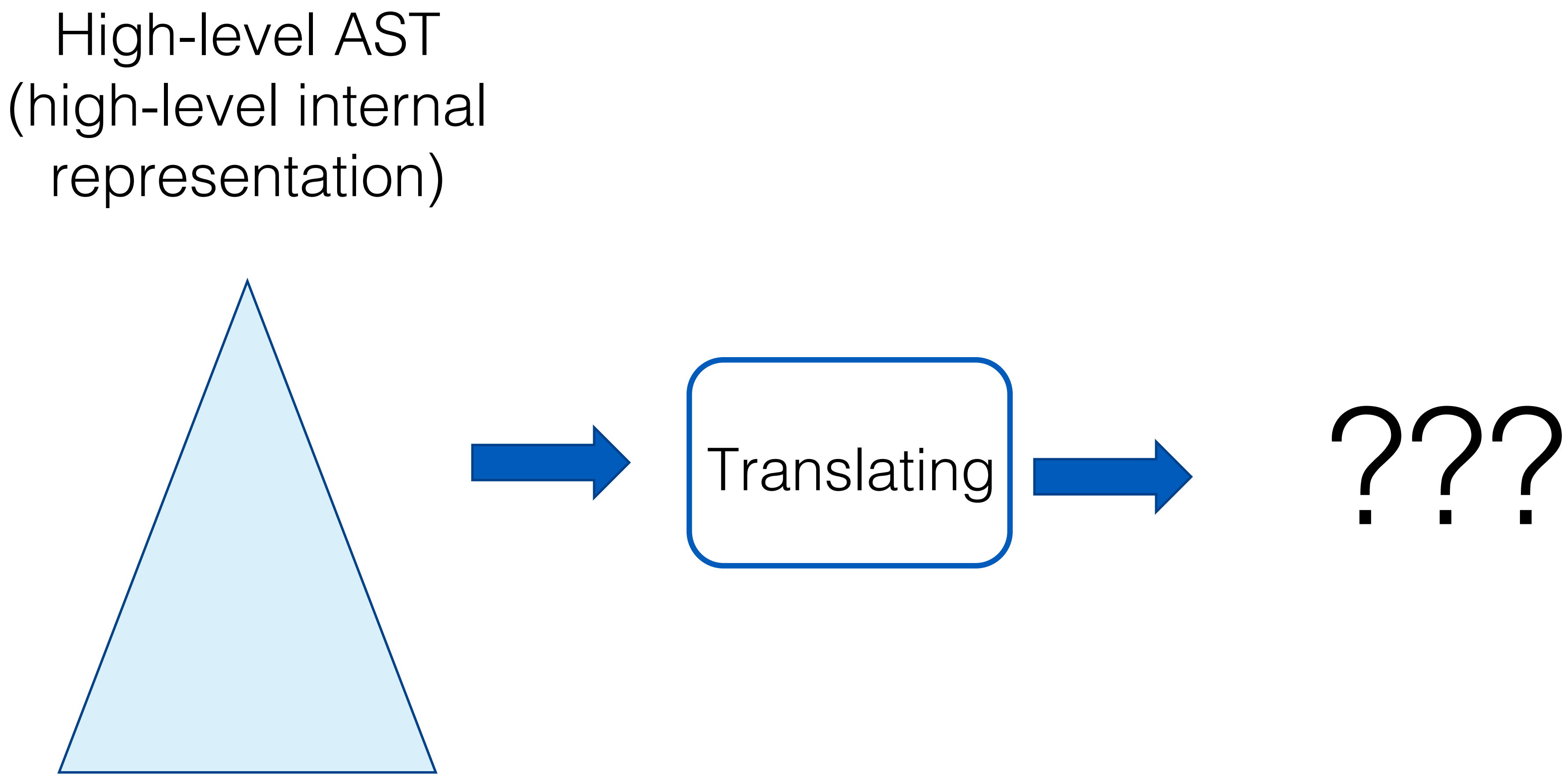


The interpreter is based on several steps as well

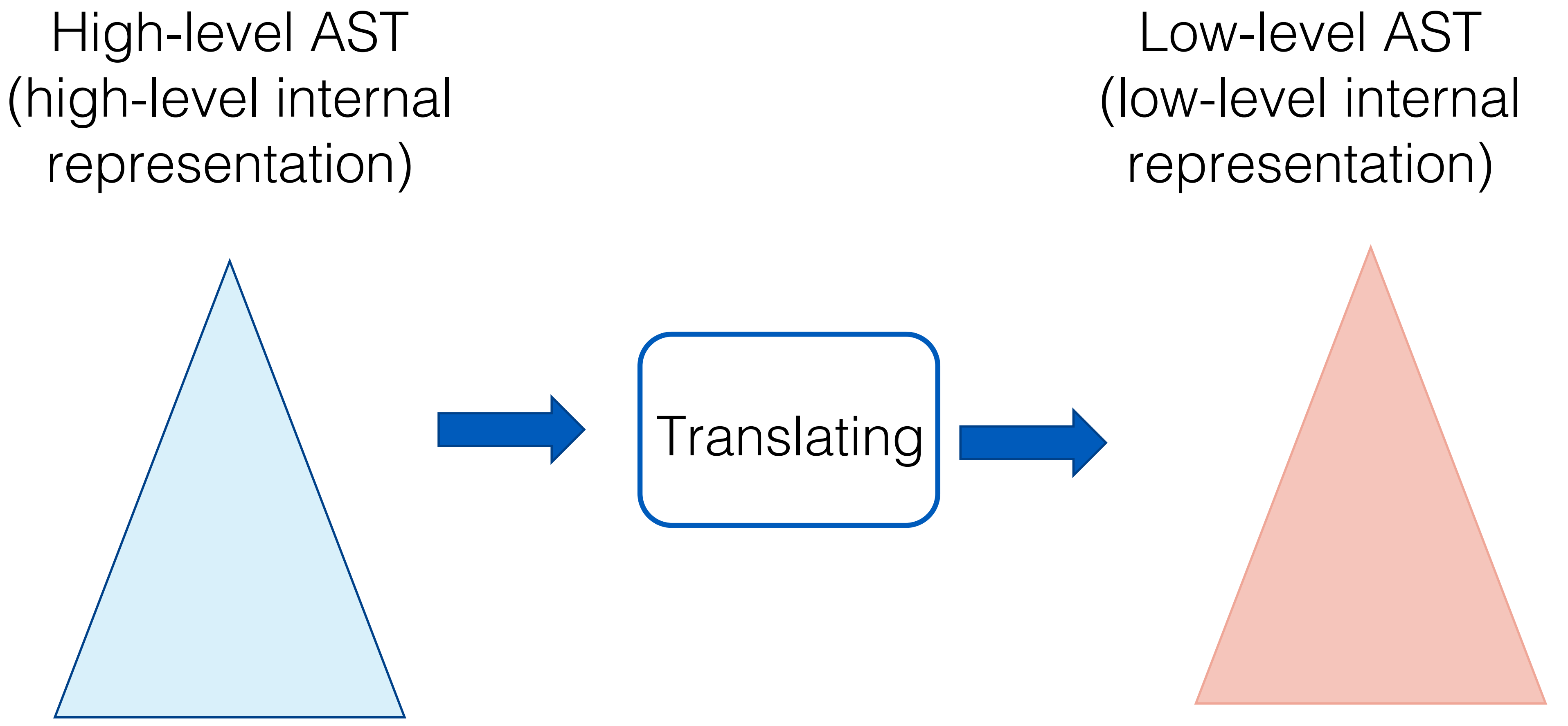
Our compiler overview



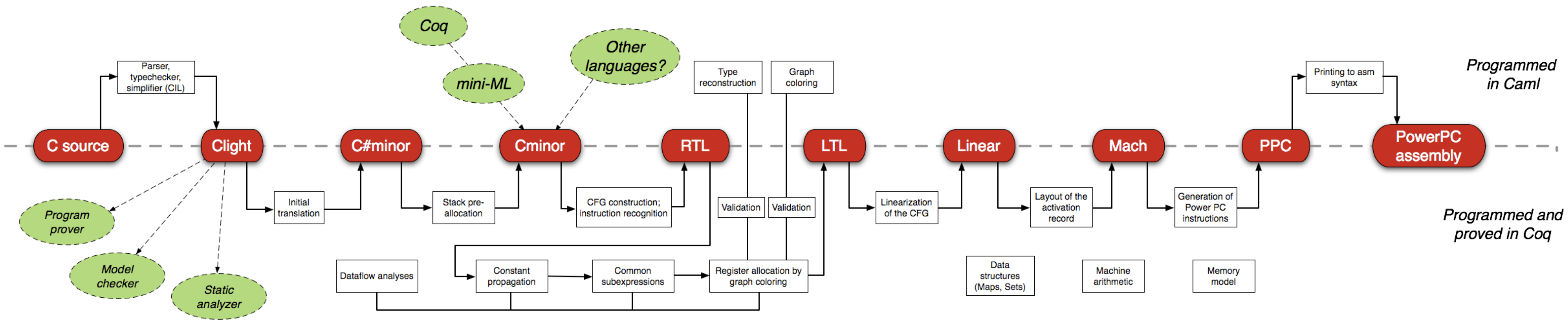
Our compiler overview



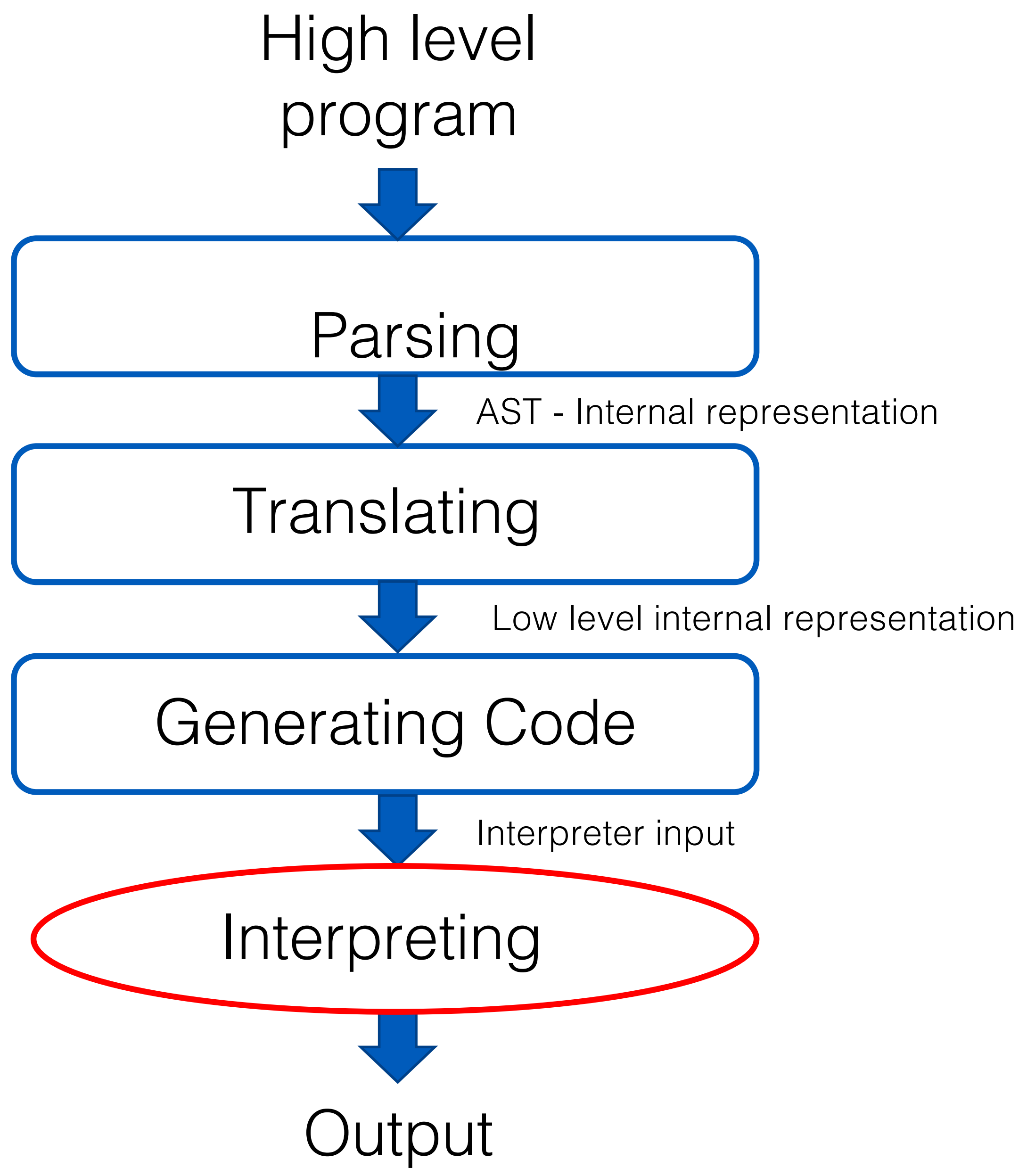
Our compiler overview


High-level AST
(high-level internal representation)
Translating
Low-level AST
(low-level internal representation)

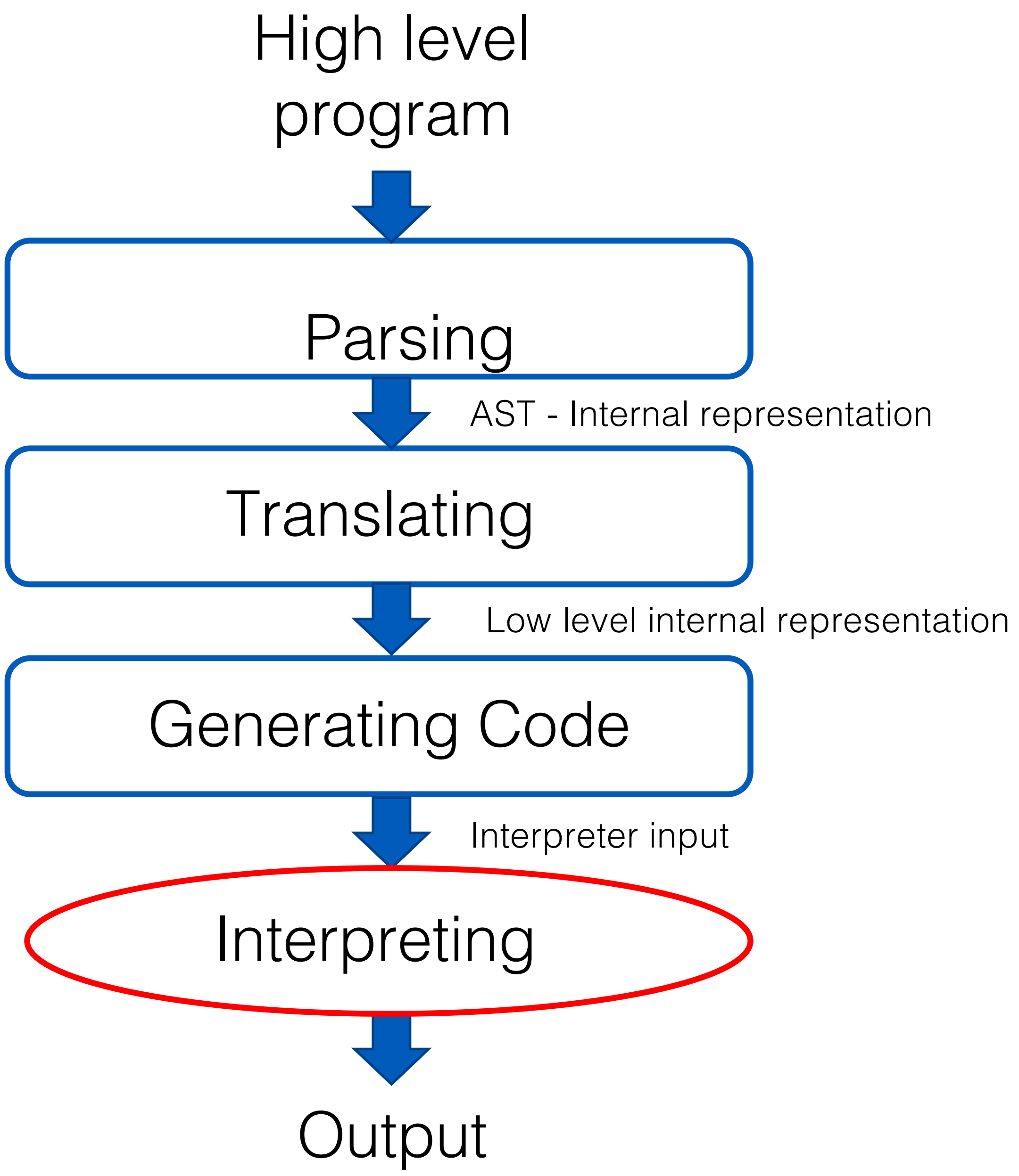
Some examples - GHC



Our compiler overview



Tip for project part3: data types



OCaml Datatypes

string

expr

cmds

string

string list

Tip for project part3: Testing your compile

$$\begin{tikzcd}
Prog_f \arrow[r, "compile"] \arrow[dr, "interp_f"] & Prog_s \arrow[d, "interp_s"] \\
 & Output
\end{tikzcd}$$

```
stack_interp (translate Progf) = func_interpreter Progf
```

Some example

How shall we proceed with the following program?

```
5-2-3
```

Some example

How shall we proceed with the following program?

```
5-2-3
```

First, the parser will give us an internal representation:

```
BOpr(Sub,BOpr(Sub,Int 5,Int 2),Int 3)
```

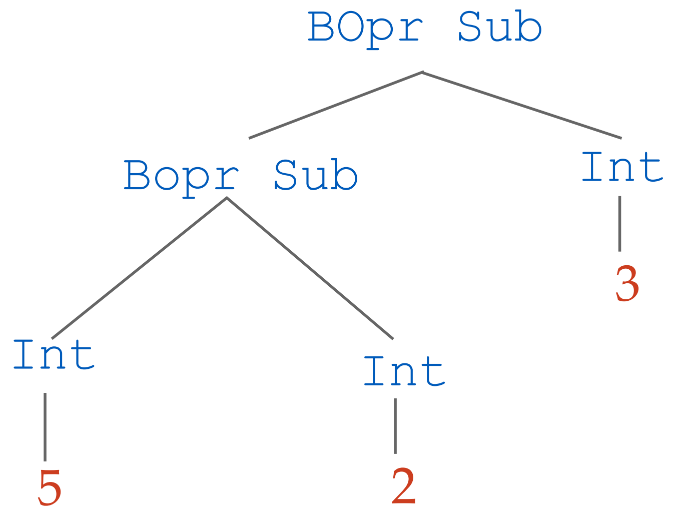
Some example

How shall we proceed with the following program?

```
5-2-3
```

First, the parser will give us an internal representation:

```
BOpr(Sub,BOpr(Sub,Int 5,Int 2),Int 3)
```



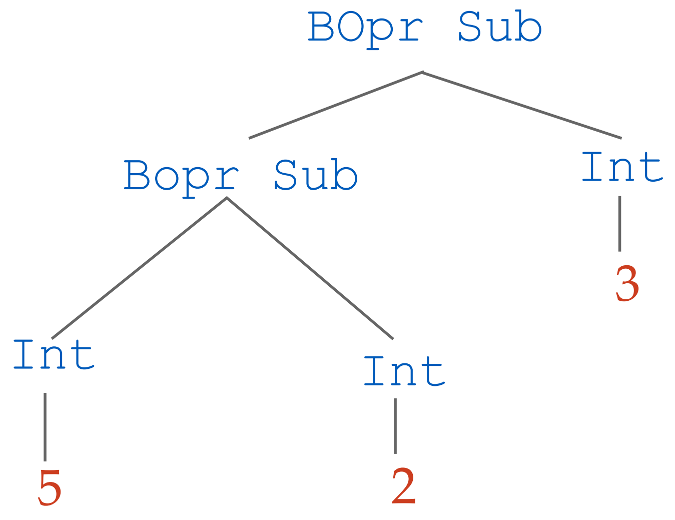
Some example

How shall we proceed with the following program?

```
5-2-3
```

First, the parser will give us an internal representation:

```
BOpr(Sub,BOpr(Sub,Int 5,Int 2),Int 3)
```



$$\textsc{SubInt}\quad \frac{v_1 \text{ and } v_2 \text{ are integers}}{[\,T\,]\ v_1 \texttt{-} v_2 \rightsquigarrow [\,T\,]\ v_1 - v_2}$$

$$\textsc{SubExpr1}\quad \frac{[\,T_1\,]\ m_1 \rightsquigarrow [\,T_2\,]\ m_2}{[\,T_1\,]\ m_1 \texttt{-} n \rightsquigarrow [\,T_2\,]\ m_2 \texttt{-} n}$$

$$\textsc{SubExpr2}\quad \frac{[\,T_1\,]\ m_1 \rightsquigarrow [\,T_2\,]\ m_2}{[\,T_1\,]\ v \texttt{-} m_1 \rightsquigarrow [\,T_2\,]\ v \texttt{-} m_2}$$

$$\textsc{SubError1}\quad \frac{v_1 \text{ or } v_2 \text{ are not integers}}{[\,T\,]\ v_1 \texttt{-} v_2 \rightsquigarrow [\,\texttt{"Panic"} :: T\,]\ \mathbf{Error}}$$

$$\textsc{SubError2}\quad \frac{[\,T_1\,]\ m \rightsquigarrow [\,T_2\,]\ \mathbf{Error}}{[\,T_1\,]\ m \texttt{-} n \rightsquigarrow [\,T_2\,]\ \mathbf{Error}}$$

$$\textsc{SubError3}\quad \frac{[\,T_1\,]\ m \rightsquigarrow [\,T_2\,]\ \mathbf{Error}}{[\,T_1\,]\ v \texttt{-} m \rightsquigarrow [\,T_2\,]\ \mathbf{Error}}$$

Some example

How shall we proceed with the following program?

```
5-2-3
```

What is our target?

Some example

How shall we proceed with the following program?

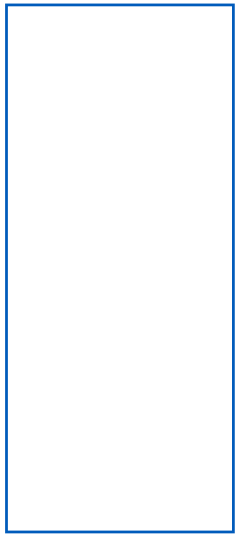
`5-2-3`

What is our target?

Program

???

Stack



Some example

How shall we proceed with the following program?

5-2-3

What is our target?

Program

```
Push 5
Push 2
Push 3
  Sub
  Sub
```

Stack

Some example

How shall we proceed with the following program?

$5 - 2 - 3$

What is our target?

Is this a good target?

Program

```
Push 5
Push 2
Push 3
  Sub
  Sub
```

Stack

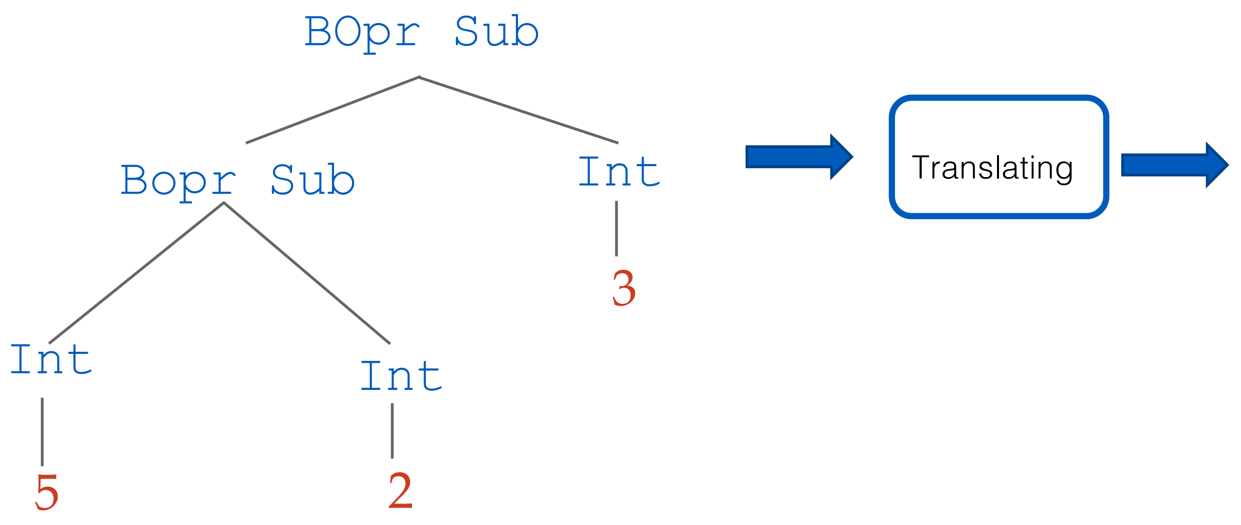
Some example

How shall we proceed with the following program?

```
5-2-3
```

What is our target?

Do they follow the same structure?



Program

```
Push 5
Push 2
Push 3
  Sub
  Sub
```

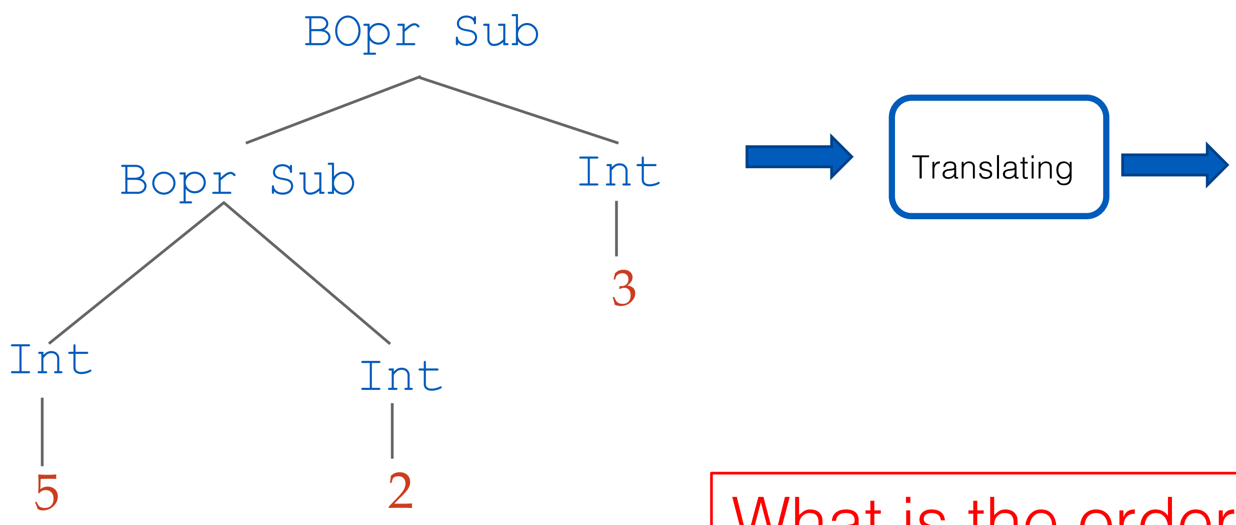
Some example

How shall we proceed with the following program?

5-2-3

What is our target?

Better solution! But it still has problems…



Program

```
Push 5
Push 2
  Sub
Push 3
  Sub
```

What is the order on the stack?

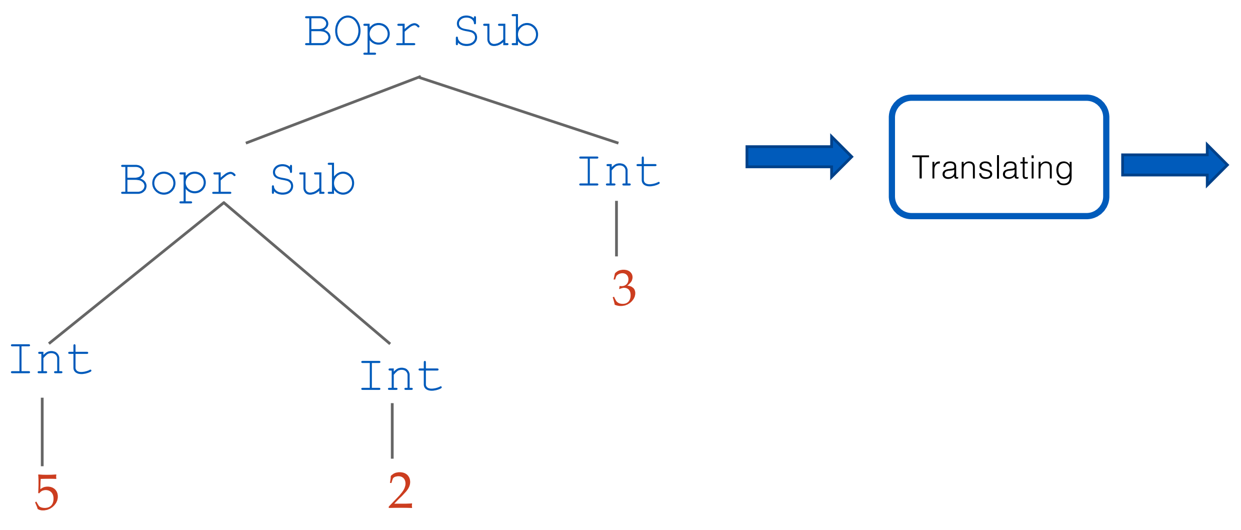
Some example

How shall we proceed with the following program?

```
5-2-3
```

What is our target?

Better solution! But it still has one problem…



Program

```
Push 5
Push 2
 Swap
 Sub
Push 3
 Swap
 Sub
```

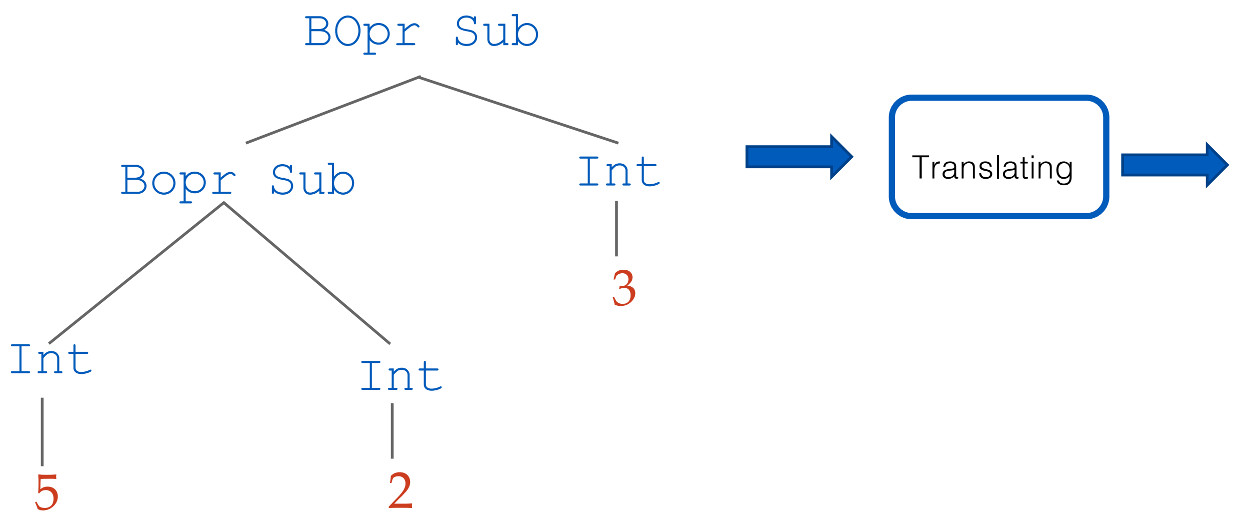
Some example

How shall we proceed with the following program?

`5-2-3`

What is our target?

This is good!



Program

```
Push (Int 5)
Push (Int 2)
    Swap
    Sub
Push (Int 3)
    Swap
    Sub
```

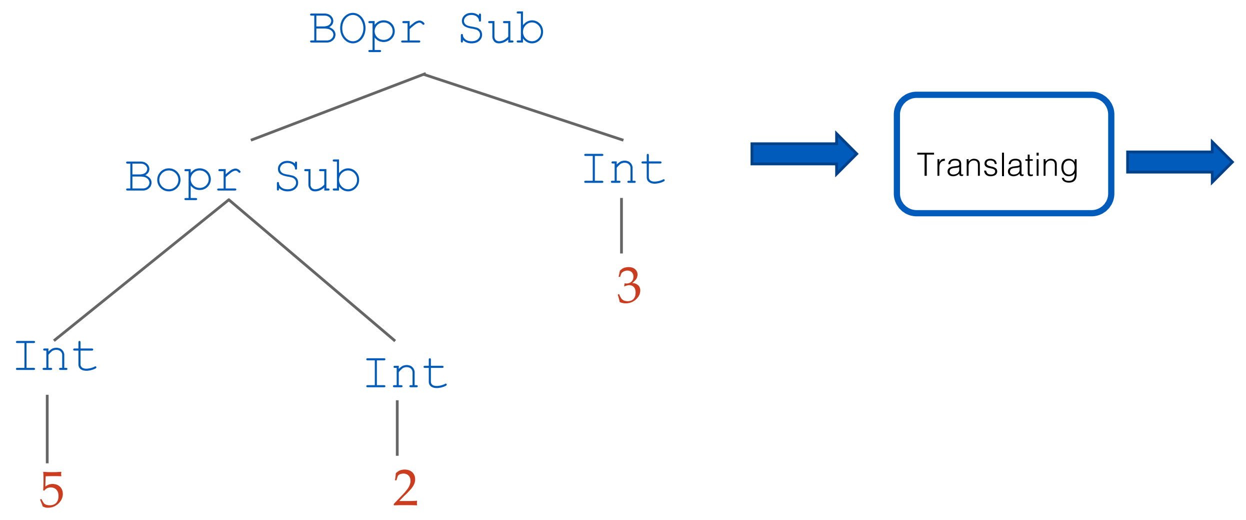
Some example

How shall we proceed with the following program?

5-2-3

What is our target?

This is good!



Program

```
Push (Int 5)
Push (Int 2)
    Swap
    Sub
Push (Int 3)
    Swap
    Sub
```

```
translate BOpr(Sub,t1,t2) = (translate t1)@(translate t2)@[Swap; Sub]
```

High-level functional language

```
…

<expr> ::= fun <var> <var> -> <expr> | <expr> <expr> |
            <expr> + <expr> | <int>
            let <var> = <expr> in <expr>

            …
```

Stack-based language

```
…

<com>    ::= Push <const> | Add |

           | Bind | Lookup

           | Fun <coms> End  | Call | Return
```

What are the conceptual differences?

Variable definition and use

High-level functional language

```
let x = 5 in x + 1
```

Stack-based language

```
???
```

Variable definition and use

High-level functional language

```
let x = 5 in x + 1
```

Stack-based language

```
Push (I 5);
Push (Sym x);
Bind;
Push (Sym x);
Lookup;
Push (I 1);
Swap;
Add
```

Variable definition and use

How do we distinguish def and use here?

High-level functional language

```
let x = 5 in x + 1
```

Stack-based language

```
Push (I 5);
Push (Sym x);
Bind;
Push (Sym x);
Lookup;
Push (I 1);
Swap;
Add
```

Variable definition and use

High-level functional language

How do we distinguish def and use here?

```
let x = 5 in x + 1
```

definition

use

Stack-based language

```
Push (I 5);
Push (Sym x);
Bind;
Push (Sym x);
Lookup;
Push (I 1);
Swap;
Add
```

Variable definition and use

High-level functional language

How do we distinguish def and use here?

```
let x = 5 in x + 1
```

definition

use

Stack-based language

```
Push (I 5);
Push (Sym x);
Bind;
Push (Sym x);
Lookup;
Push (I 1);
Swap;
Add
```

definition

use

Variable definition and use

High-level functional language

How do we distinguish def and use here?

```
let x = 5 in x + 1
```

definition

use

Stack-based language

```
Push (I 5);
Push (Sym x);
Bind;
Push (Sym x);
Lookup;
Push (I 1);
Swap;
Add
```

definition

use

How do we distinguish them?

Variable definition and use

High-level functional language

```
let x = <expr> in <expr>
```

definition

use

Stack-based language

```
???
```

```
translate Let(x,t1,t2) = ??
```

How do we generalize?

Function definition and function call

High-level functional language

```
let rec fact x =
if x <= 0 then 1
else x * fact (x - 1)
in trace (fact 10)
```

Stack-based language

```
???
```

Function definition and function call

High-level functional language

```
let rec fact x =
if x <= 0 then 1
else x * fact (x - 1)
in trace (fact 10)
```

Stack-based language

```
???
```

How do we distinguish def and use here?

Function definition and function call

High-level functional language

```
let rec fact x =
if x <= 0 then 1
else x * fact (x - 1)
in trace (fact 10)
```

Function definition

Function Call

Stack-based language

```
???
```

How do we distinguish def and use here?

Function definition and function call

High-level functional language

```
let rec fact x =
if x <= 0 then 1
else x * fact (x - 1)
in trace (fact 10)
```

Function definition

Function Call

Application

Stack-based language

```
???
```

How do we distinguish def and use here?

```
translate Fun(f,x,t) = ??

translate App(t1,t2) = ??
```