Lecture 2: Compilation and Interpretation

COMP 524 Programming Language Concepts
Stephen Olivier
January 15, 2009

Based on notes by A. Block, N. Fisher, F. Hernandez-Campos, and D. Stotts

The University of North Carolina at Chapel Hill

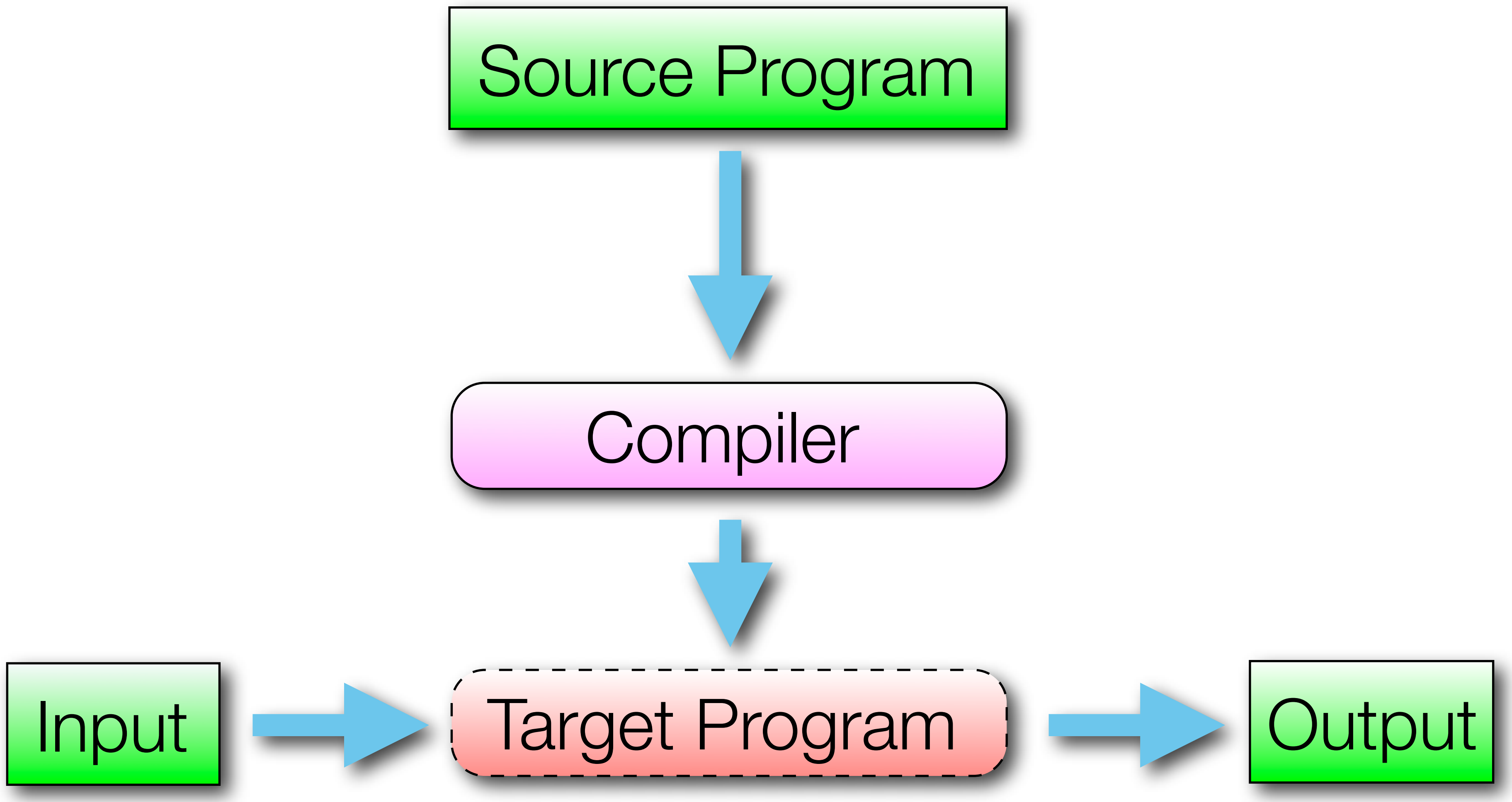
Goals

- In this lecture, we will discuss how to **translate source code** into **machine executable code**.

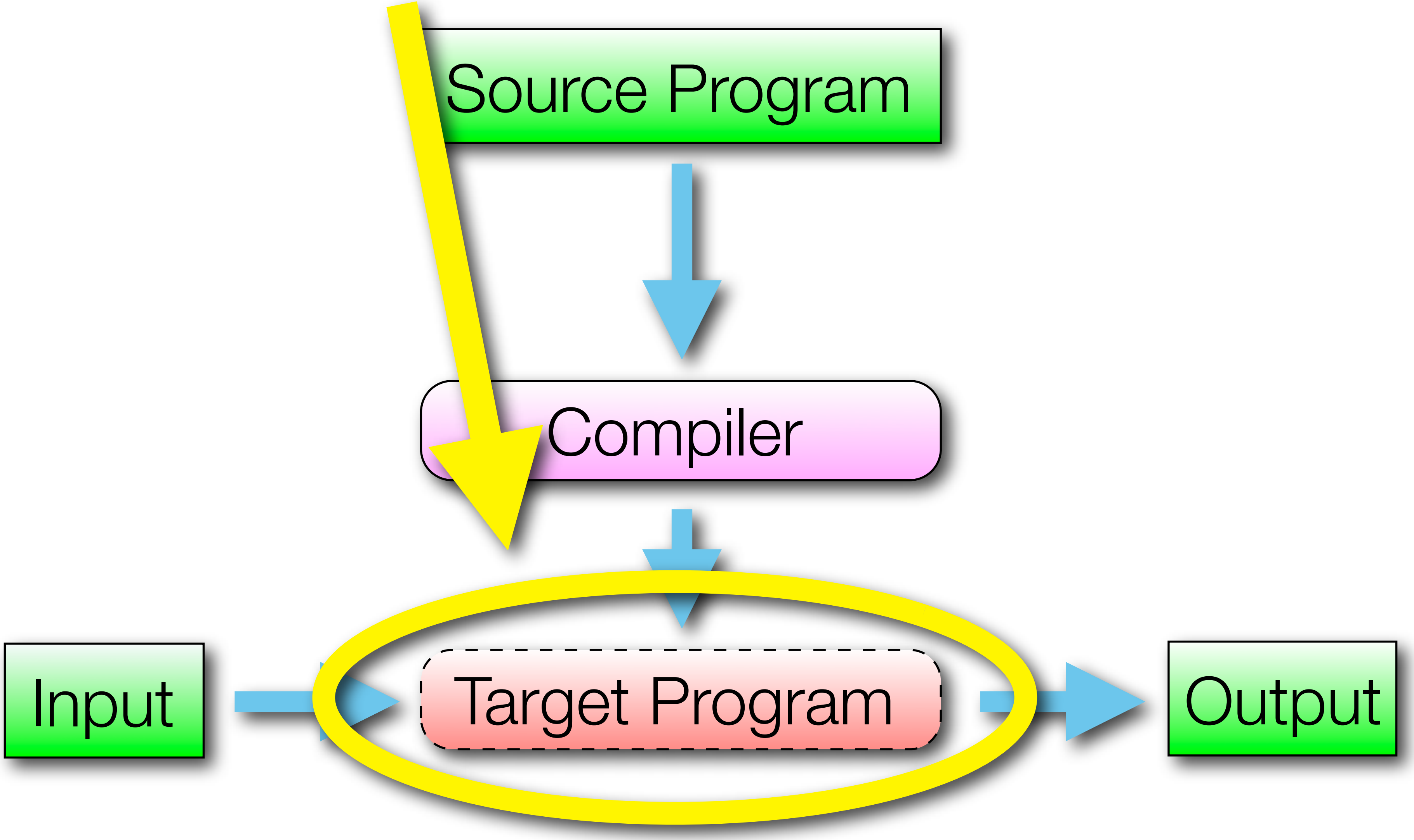
Compilation and Interpretation

- There are two primary methods for translating high-level code:
  - Compilation
  - Interpretation

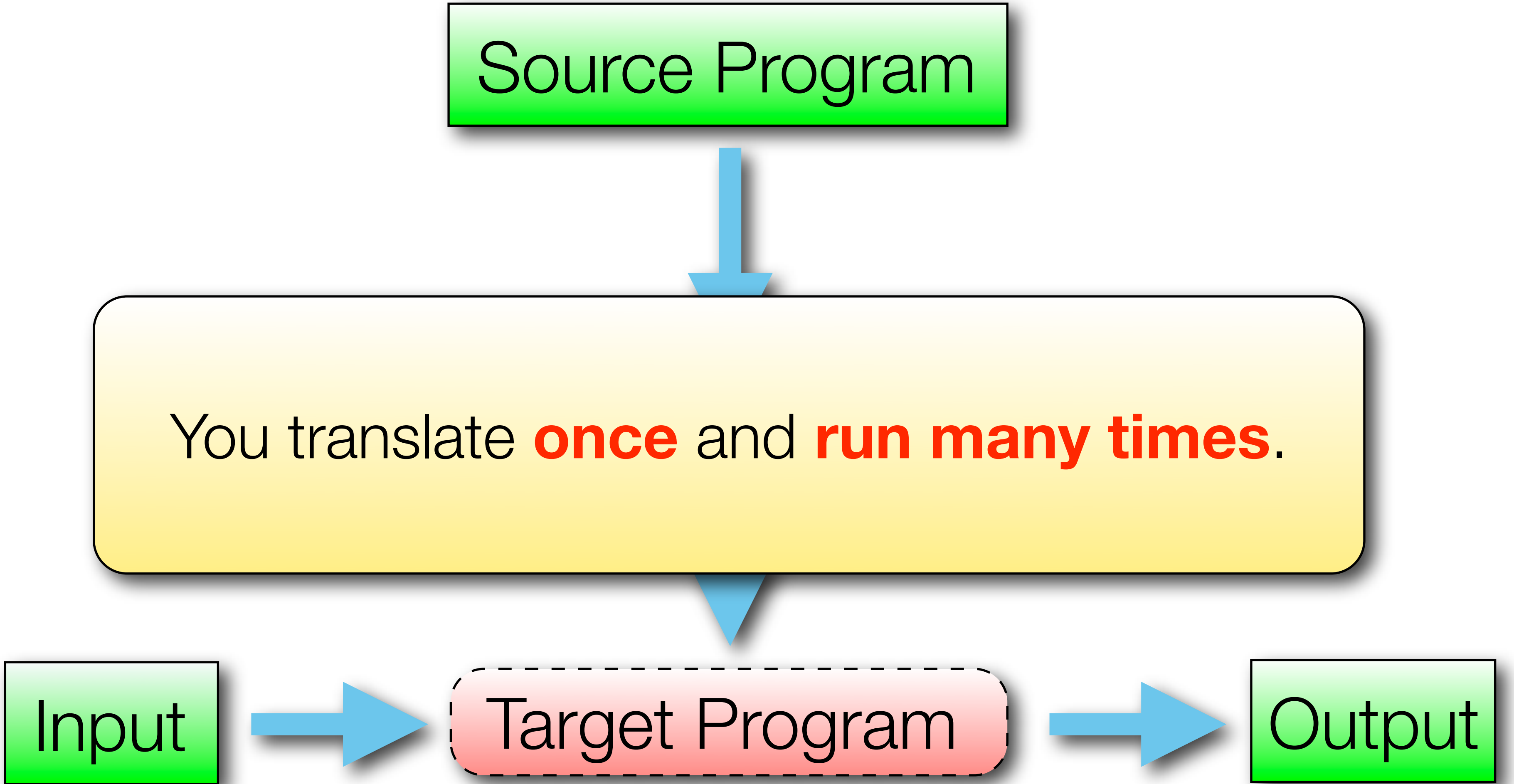
Compilation



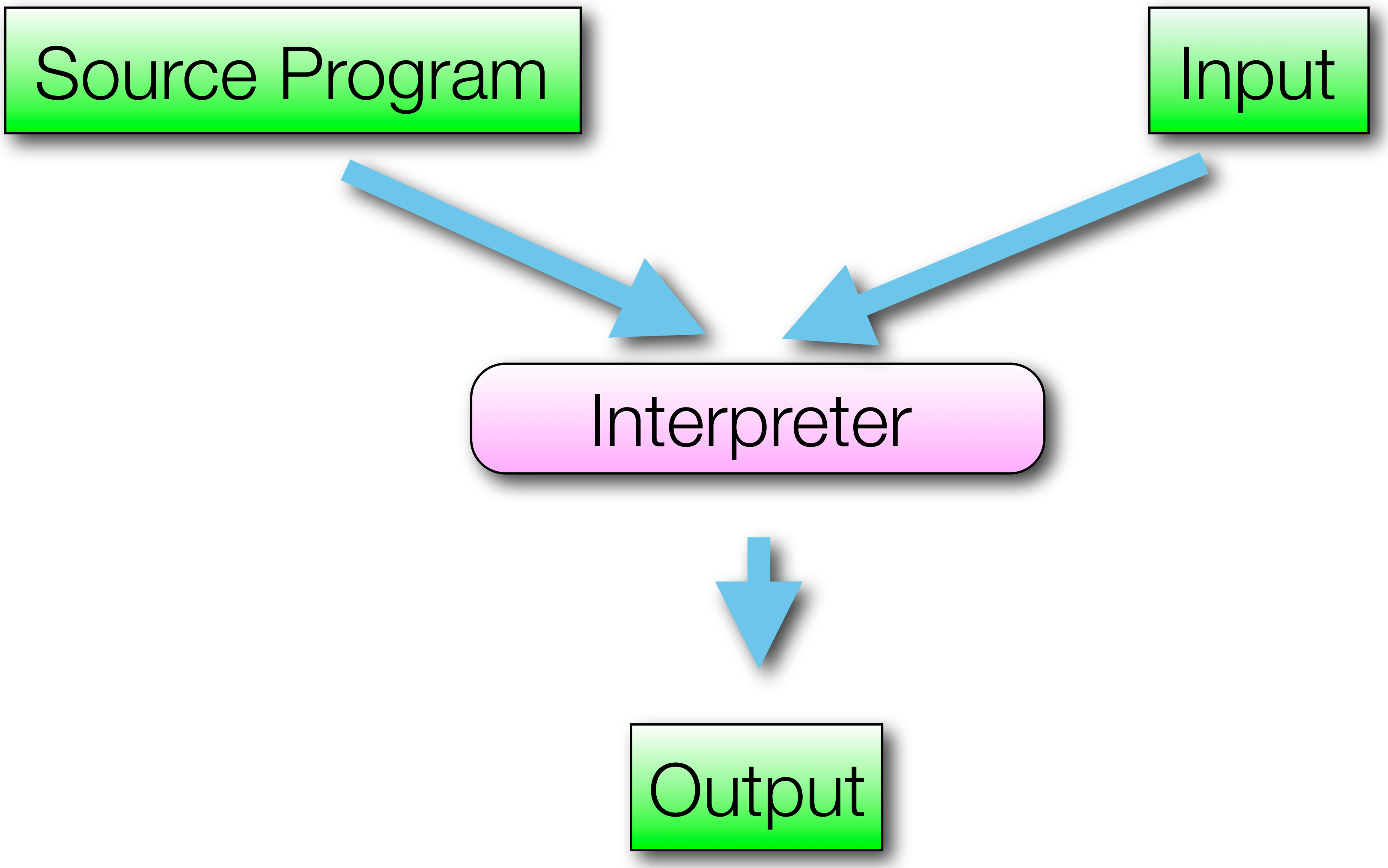
The “**target program**” is called the **object code**.



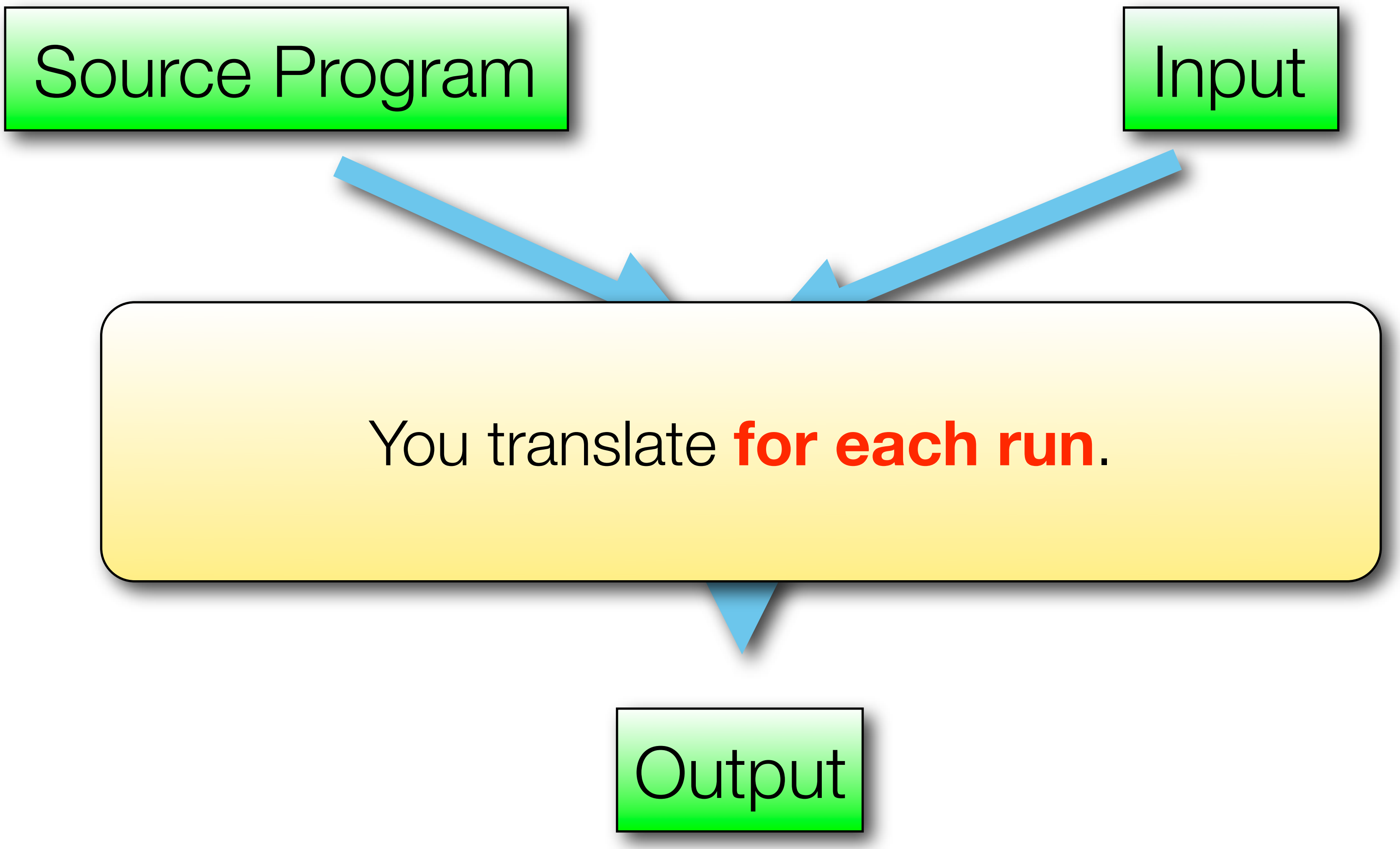
Compilation



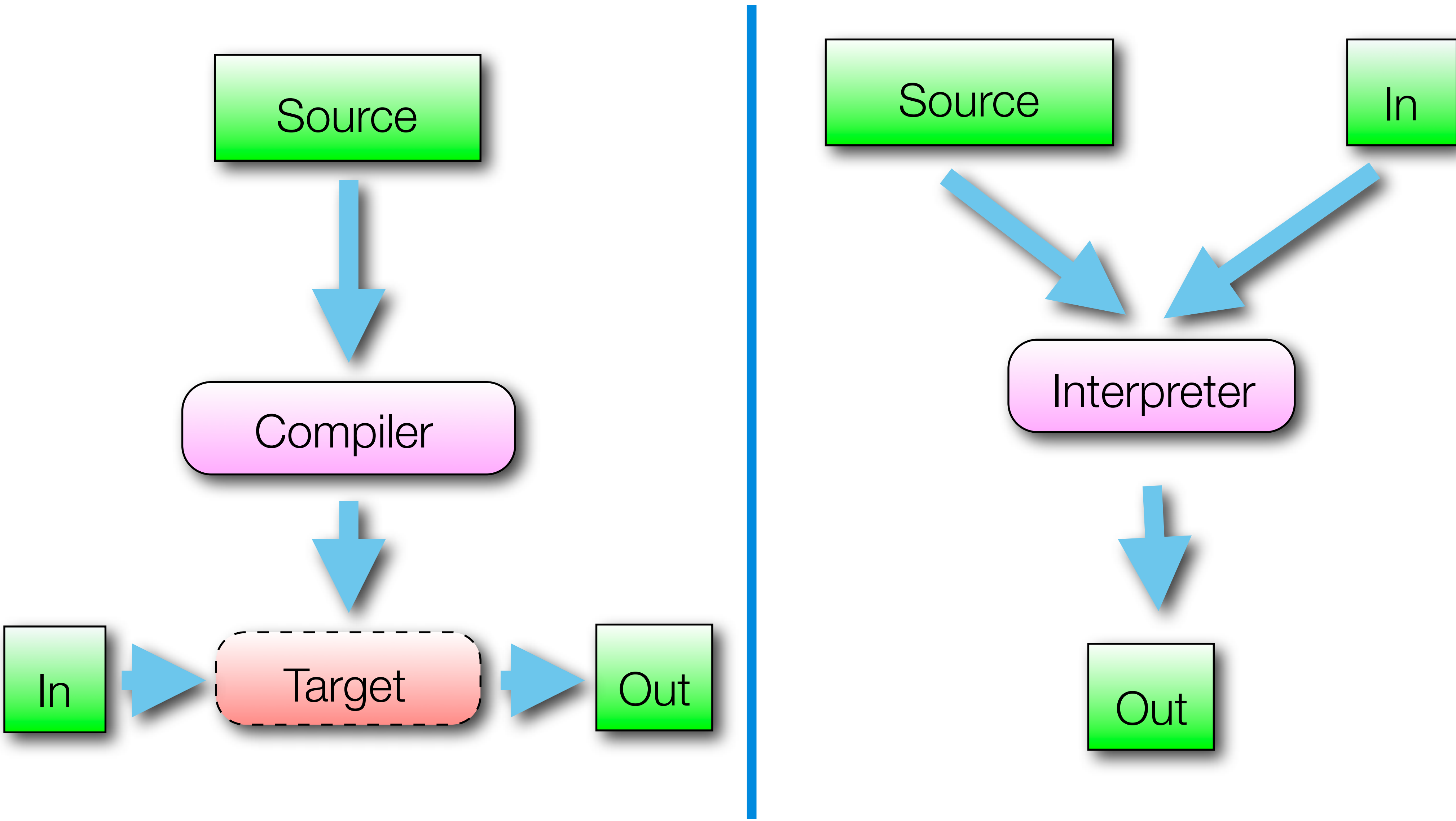
Interpretation



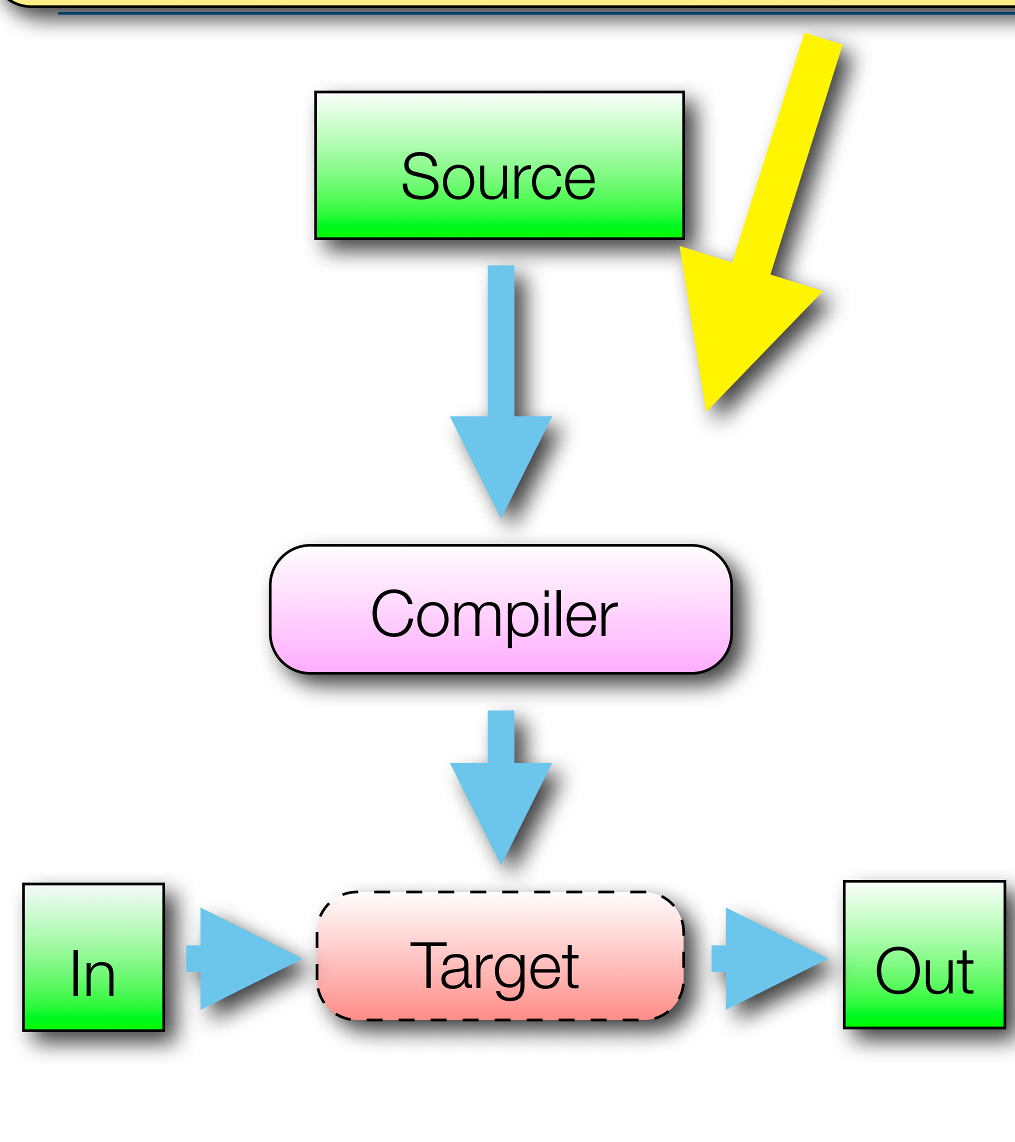
Interpretation

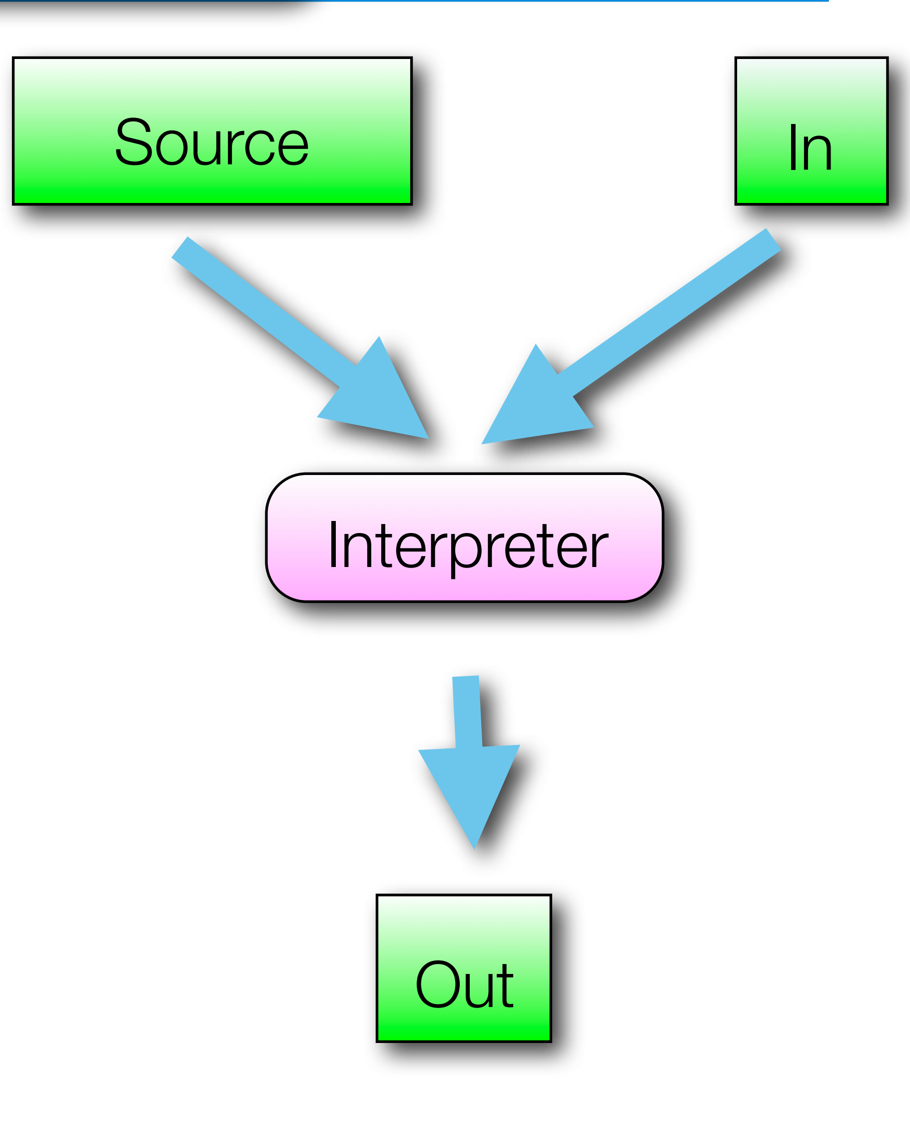


Comparison



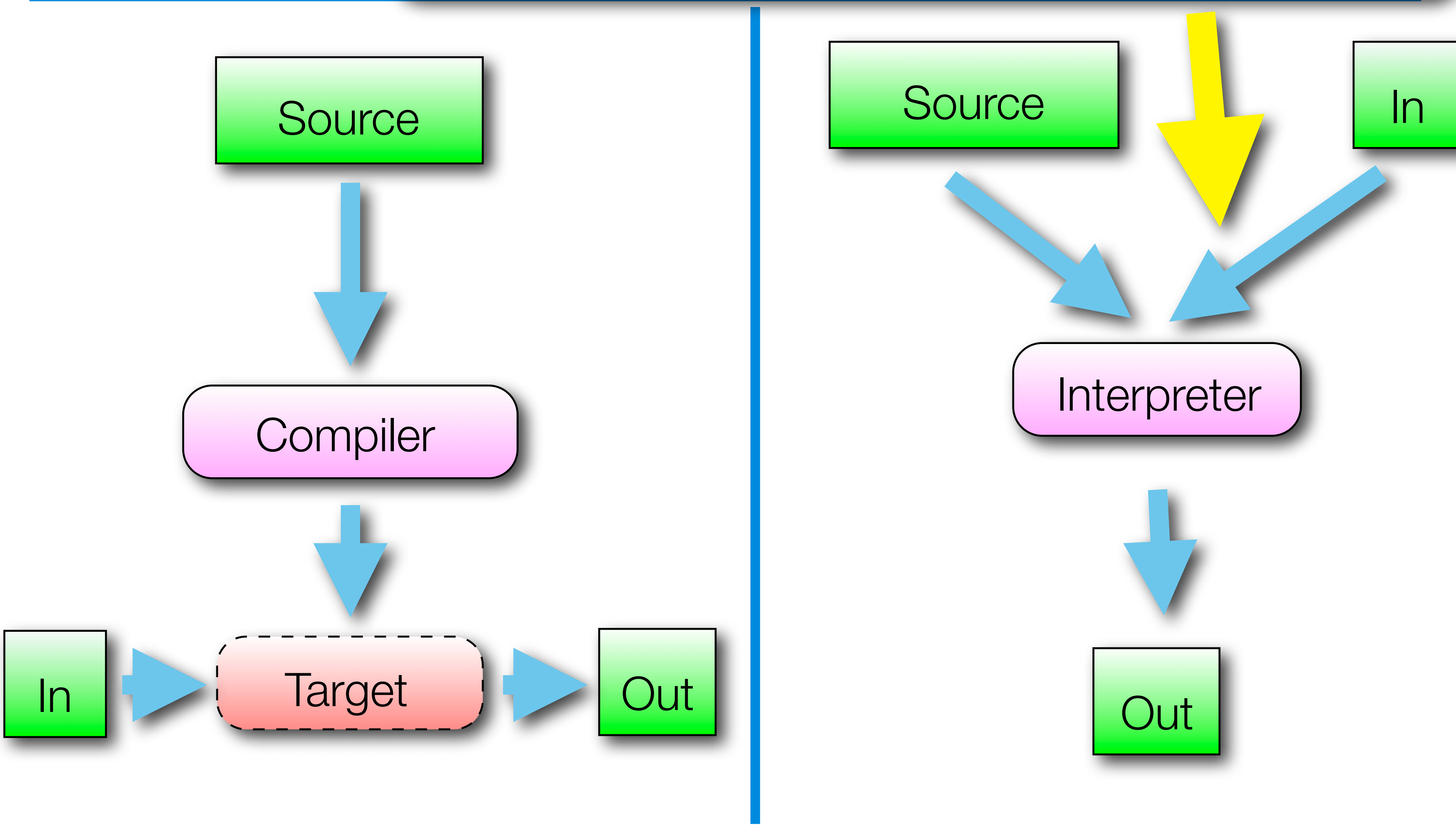
Compilers are usually **faster** than interpreters.



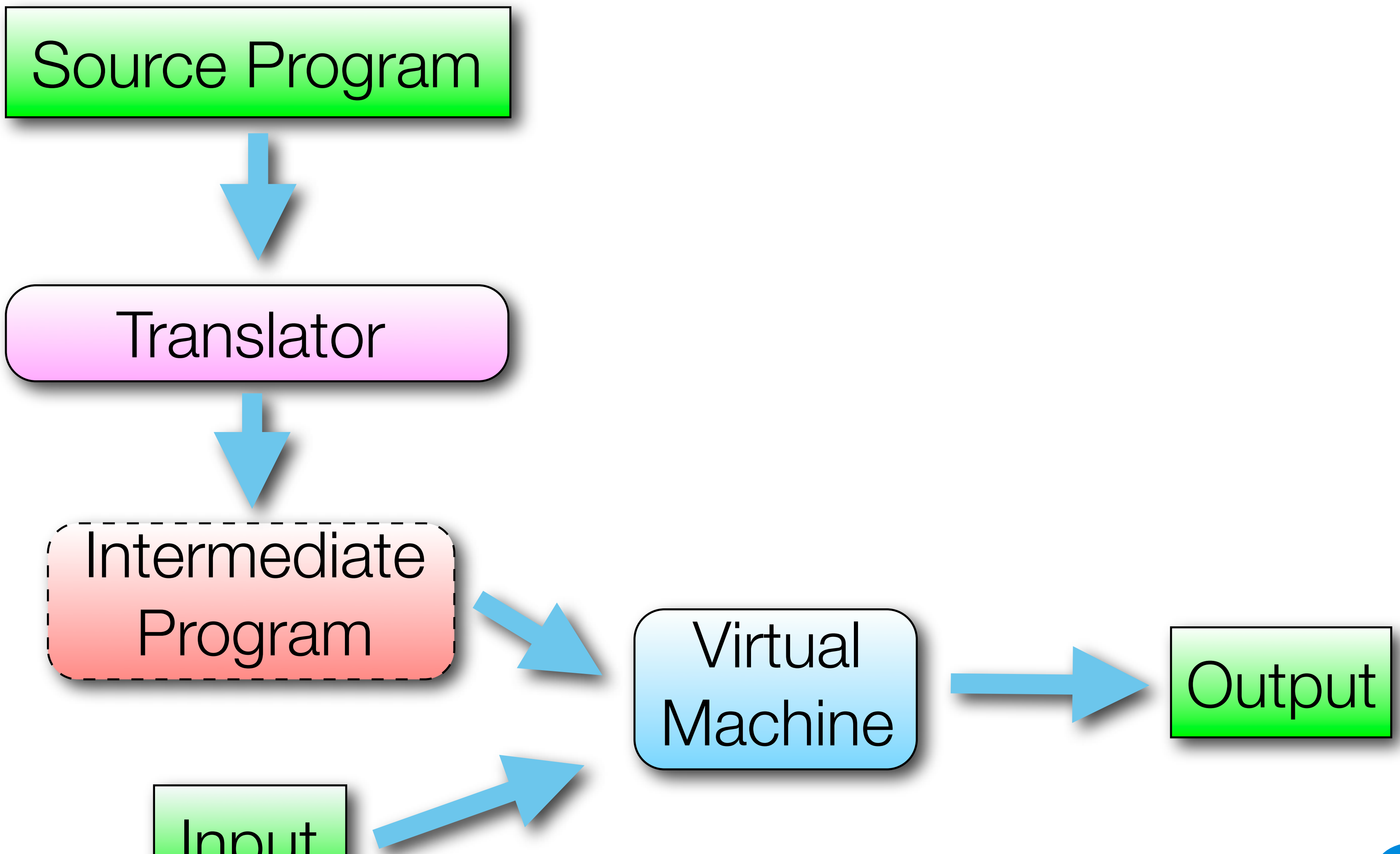


Comparison

Interpreters are usually **more flexible** and **easier to debug** than compilers.

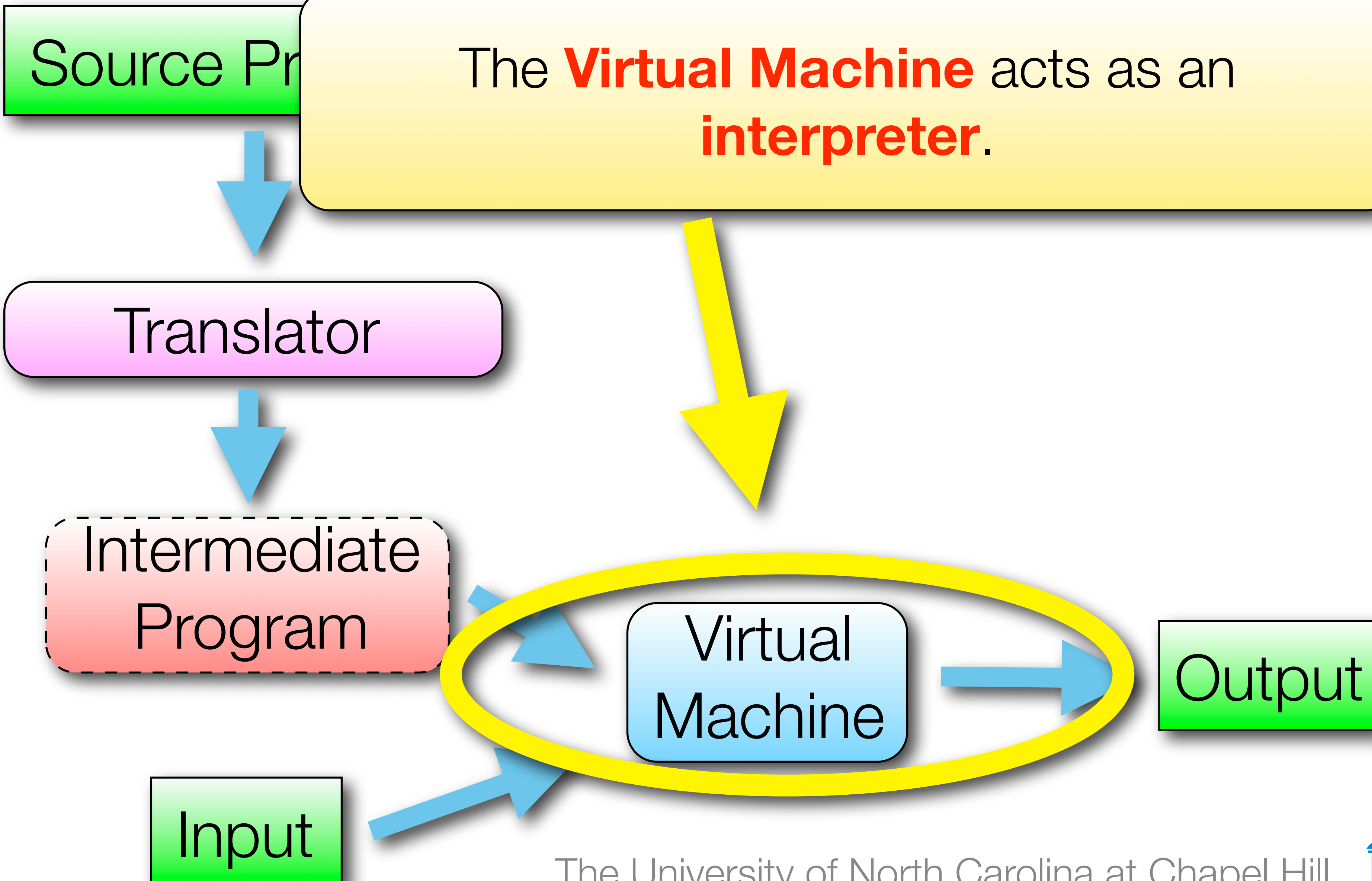


Compilation and Interpretation



Compilation and Interpretation

The **Virtual Machine** acts as an **interpreter**.

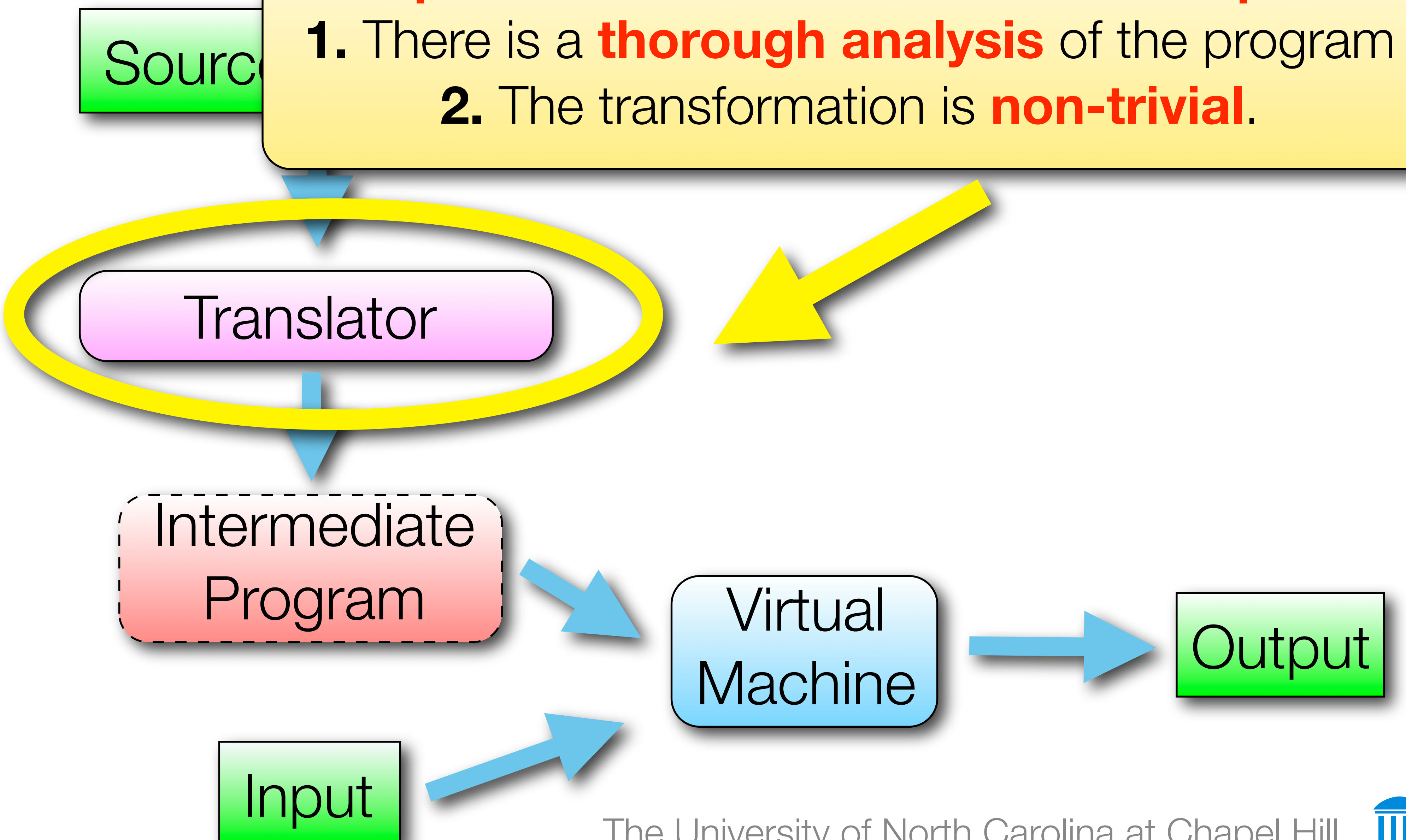


Compila

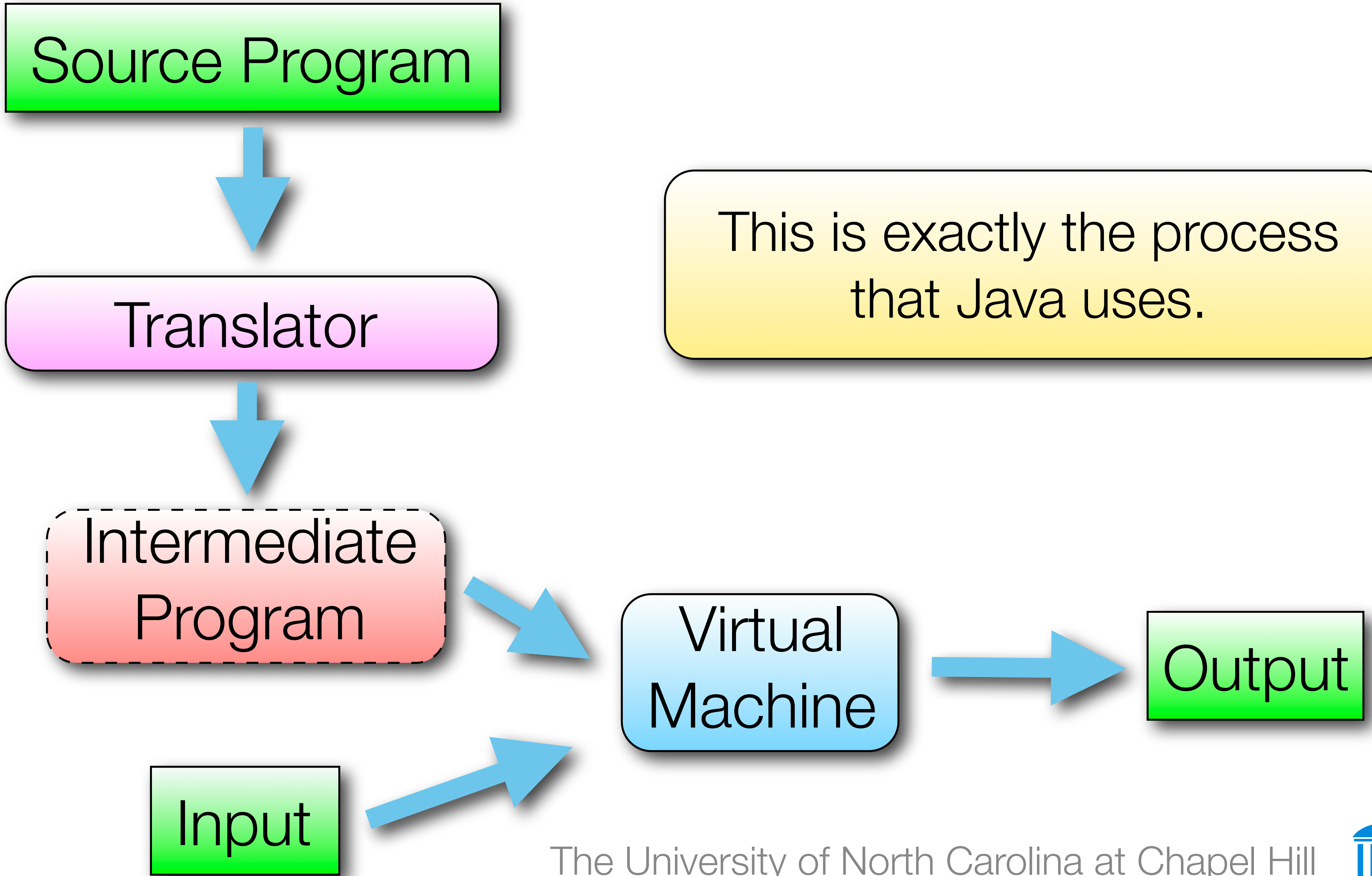
The translator can be a **compiler** or an **interpreter**. It is considered to be a **compiler** if:

1. There is a **thorough analysis** of the program

2. The transformation is **non-trivial**.

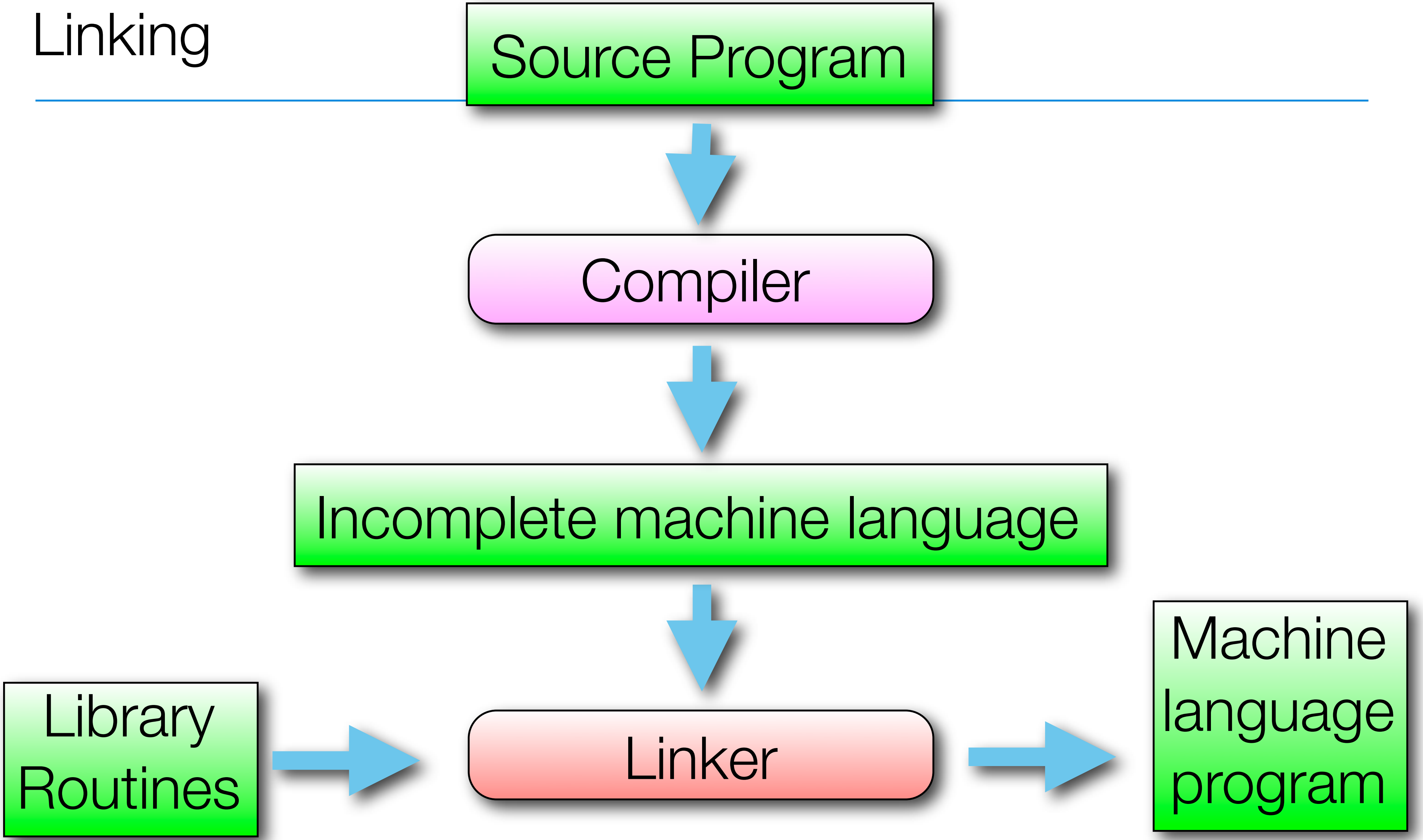


Compilation and Interpretation

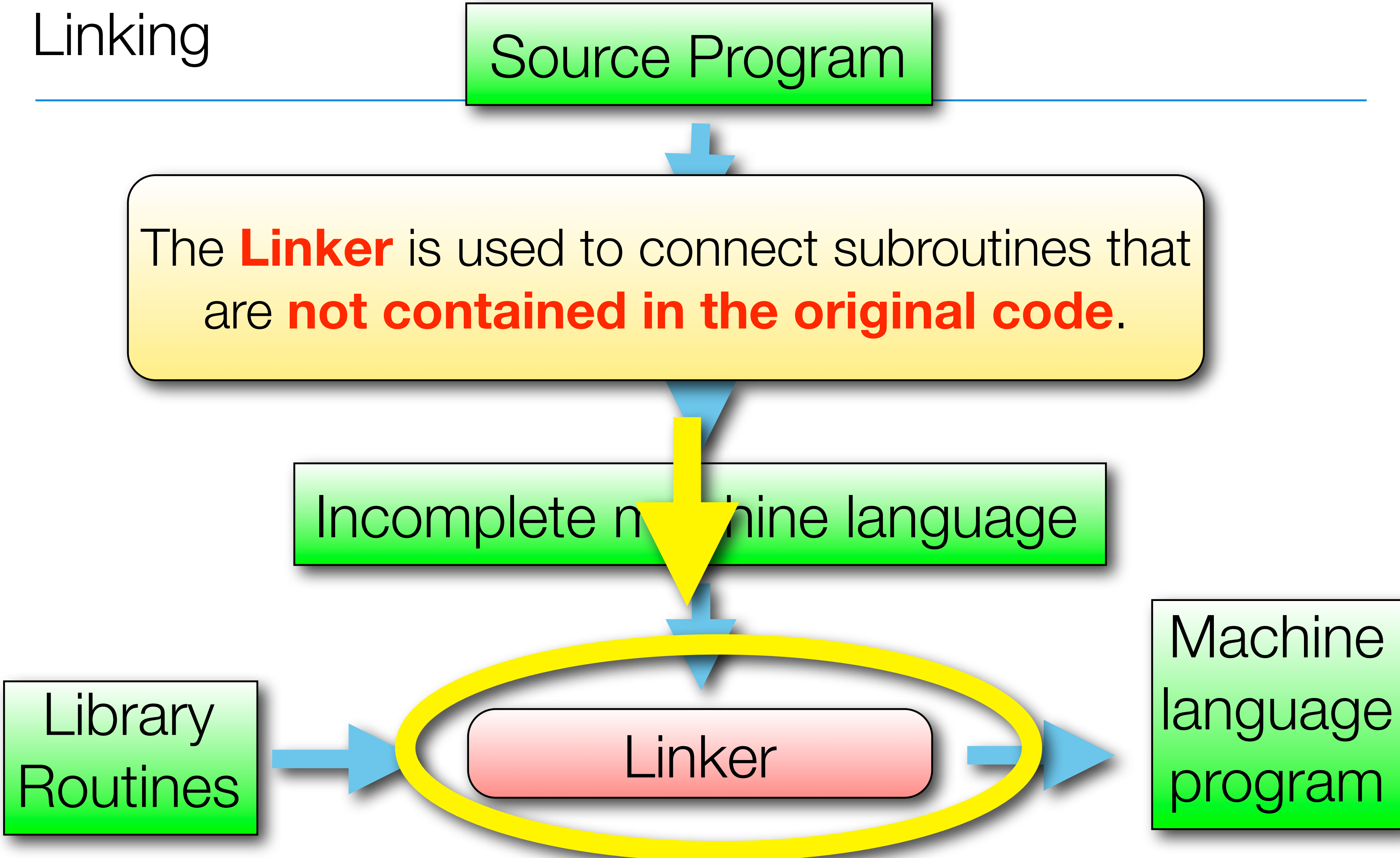


This is exactly the process that Java uses.

Linking

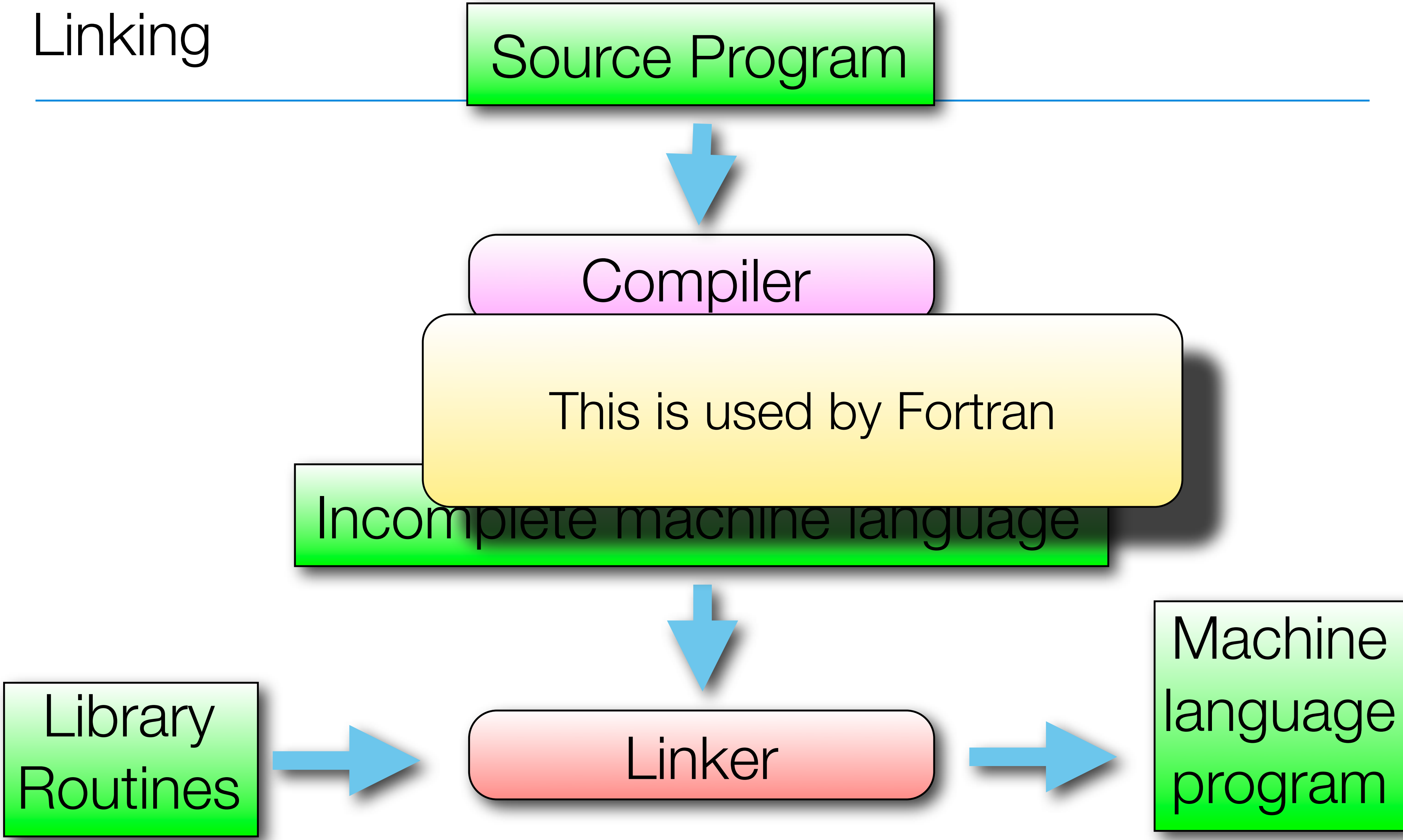

Source Program
Compiler
Incomplete machine language
Library Routines
Linker
Machine language program

Linking

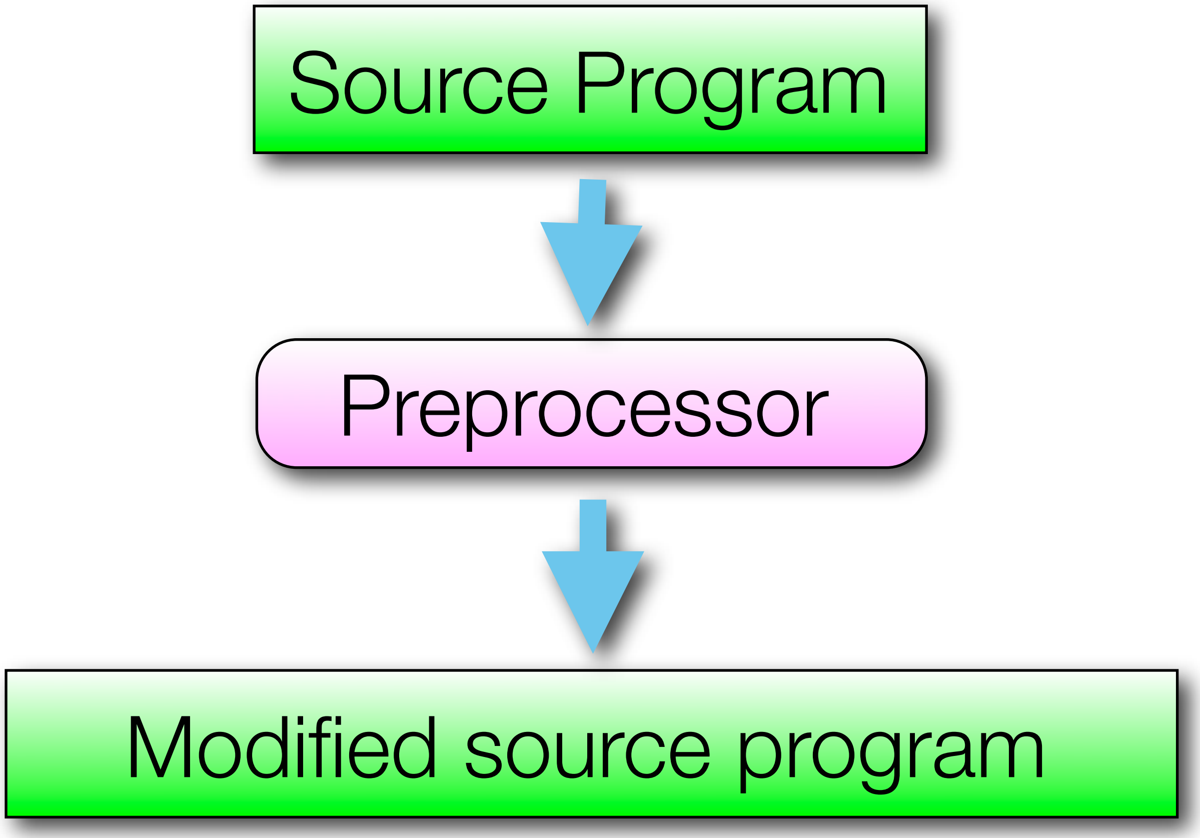


Source Program
The **Linker** is used to connect subroutines that are **not contained in the original code**.
Incomplete m hine language
Library Routines
Linker
Machine language program

Linking


Source Program
Compiler
This is used by Fortran
Incomplete machine language
Library Routines
Linker
Machine language program

Preprocessing



Preprocessing

For example, this may be used to **remove comments** or use replacement **macros**.

Preprocessor

↓

Modified source program

Preprocessing

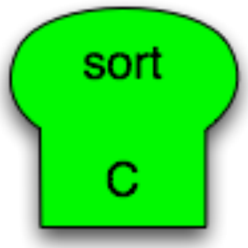
Source Program

On the note about macros, consider the macro **`#define FALSE 0`**.
This code will replace all instances of the word "**`FALSE`**" with the value **`0`**.

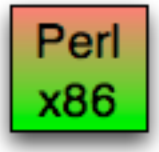
Modified source program

Tombstone Diagrams

- Useful for graphically representing programs and translators.
  - Program (in C):

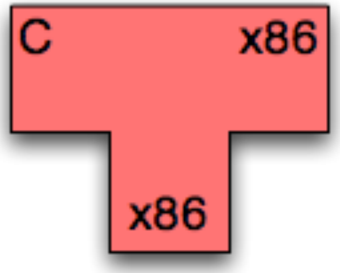


  - Interpreter (for Perl, impl. in x86)



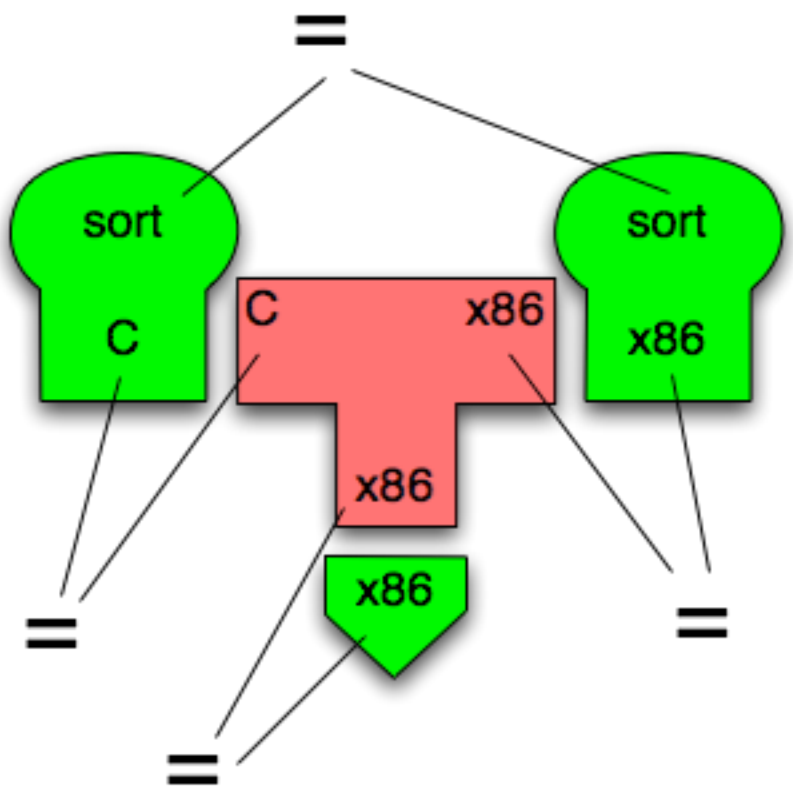
  - Machine (x86):

  - Translator (C to x86, impl. in x86)



Tombstone Diagrams

- Example: Compiling a C program to run on x86

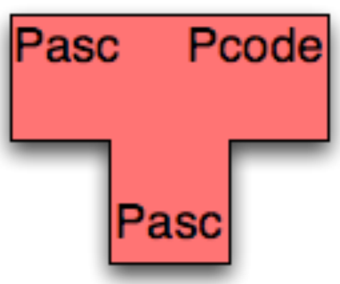


Bootstrapping

- Pascal came shipped with three things:
  - A Pascal to P-Code Compiler, written in Pascal.
  - A P-code interpreter, in Pascal.
  - A Pascal to P-Code Compiler, written in P-Code.

Bootstrapping

- Pascal came shipped with three things:
  - A Pascal to P-Code Compiler, written in Pascal.
  - A P-code interpreter, in Pascal.
  - A Pascal to P-Code Compiler, written in P-Code.



P-Code is a very simple language that can easily be translated into any machine language.

Bootstrapping

- Pascal came shipped with three things:
  - A Pascal to P-Code Compiler, written in Pascal.
  - A P-code interpreter, in Pascal.
  - A Pascal to P-Code Compiler, written in P-Code.

Bootstrapping

- Pascal came shipped with three things:
    - A Pascal to P-Code Compiler, written in Pascal.

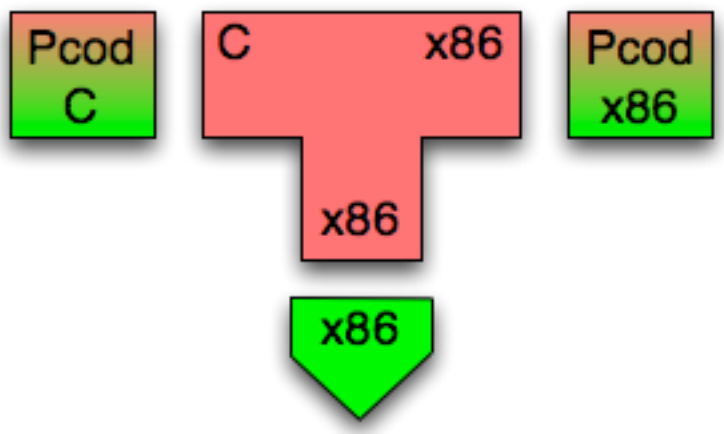
The interpreter is **hand translated** into a supported language.

    - A Pascal to P-Code Compiler, written in P-Code.

Bootstrapping

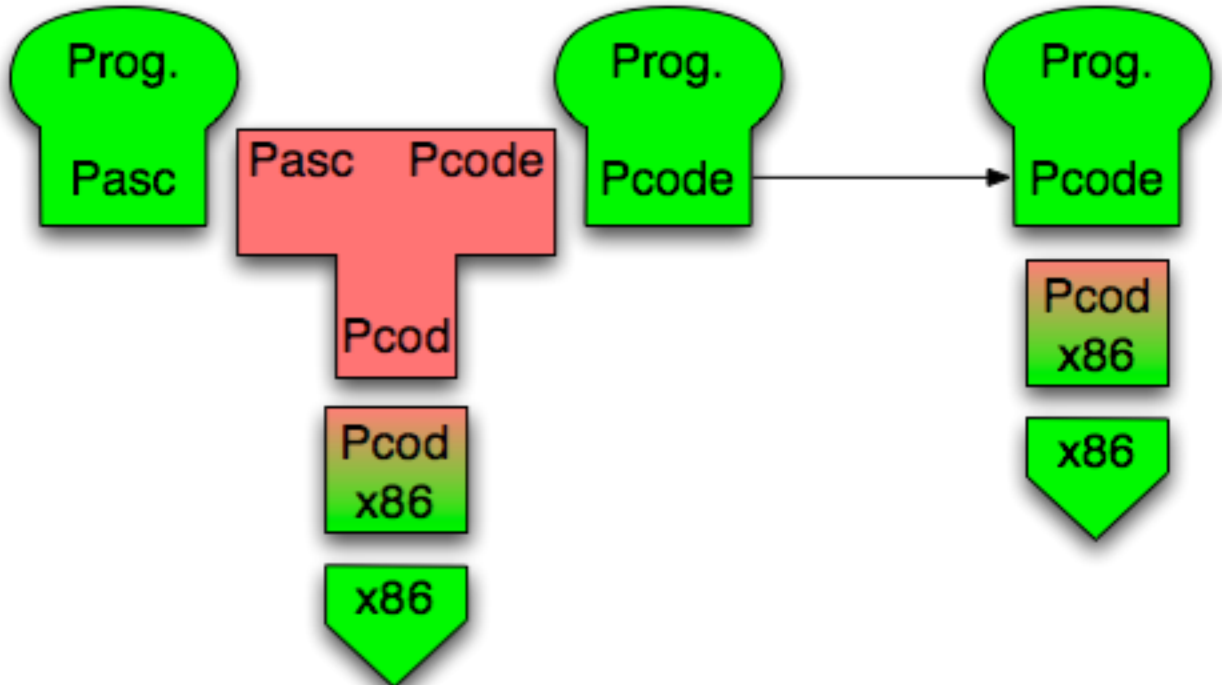
- Pascal came shipped with three things:
  - A Pascal to P-Code Compiler, written in Pascal.

The interpreter is **hand translated** into a supported language.
And then compiled into machine code.



Bootstrapping

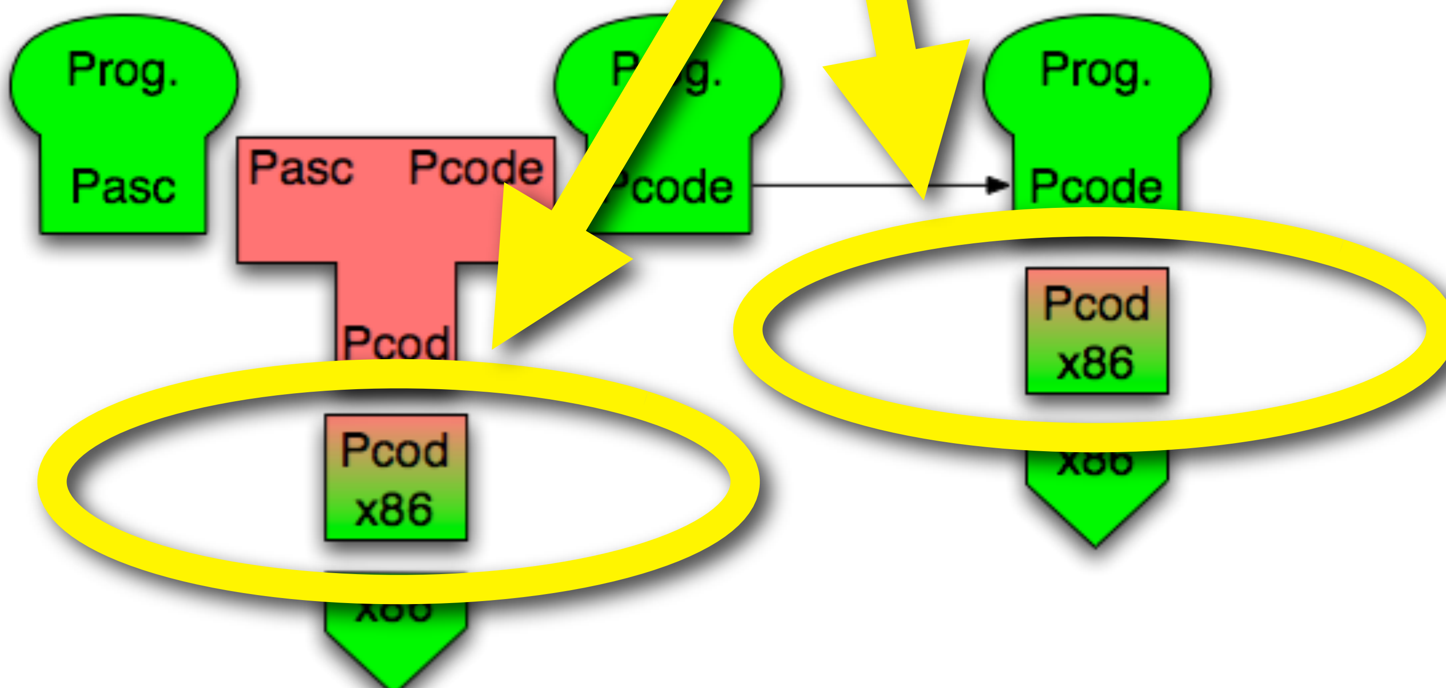
- To get a simple (but slow) compiler, one could use the “Pascal to P-Code compiler, in **P-Code**” and the “P-Code to machine lang. **interpreter**” to compile Pascal code.



Bootstrapping

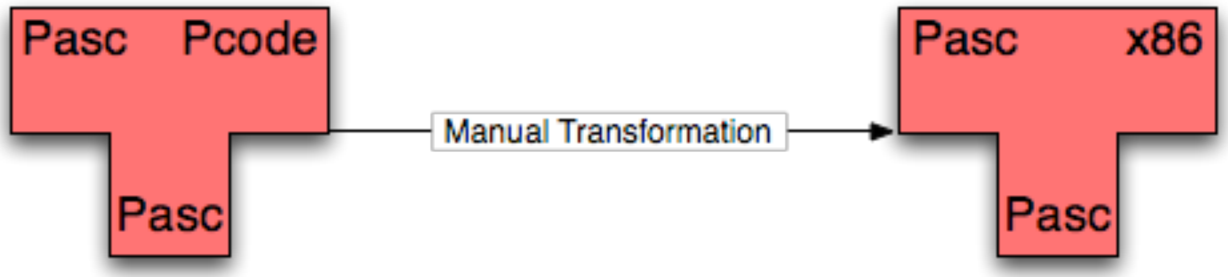
This interpreter is used to run the **Pascal to P-Code** compiler and the program.

- To get a simple (…) “Pascal to P-Code compiler, in **P-Code**” and the “P-Code to machine lang. **interpreter**” to compile Pascal code.



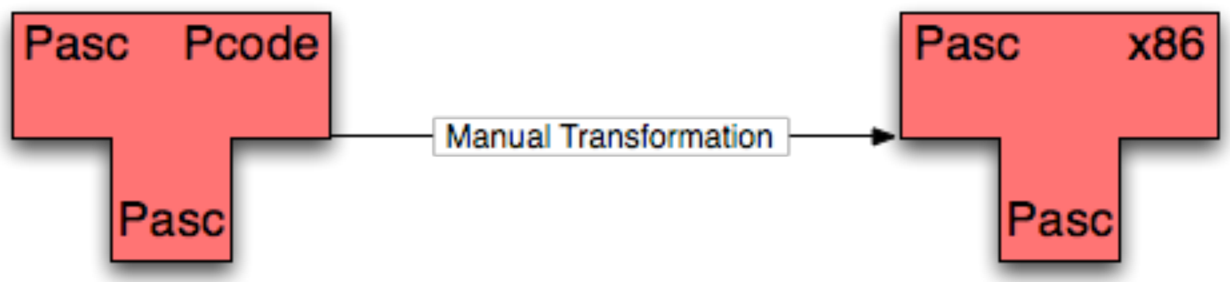
Bootstrapping (Faster)

- To create a faster compiler, we modify the “Pascal to P-Code compiler, in **Pascal**” so that it is a “Pascal to machine language compiler, in **Pascal**.”

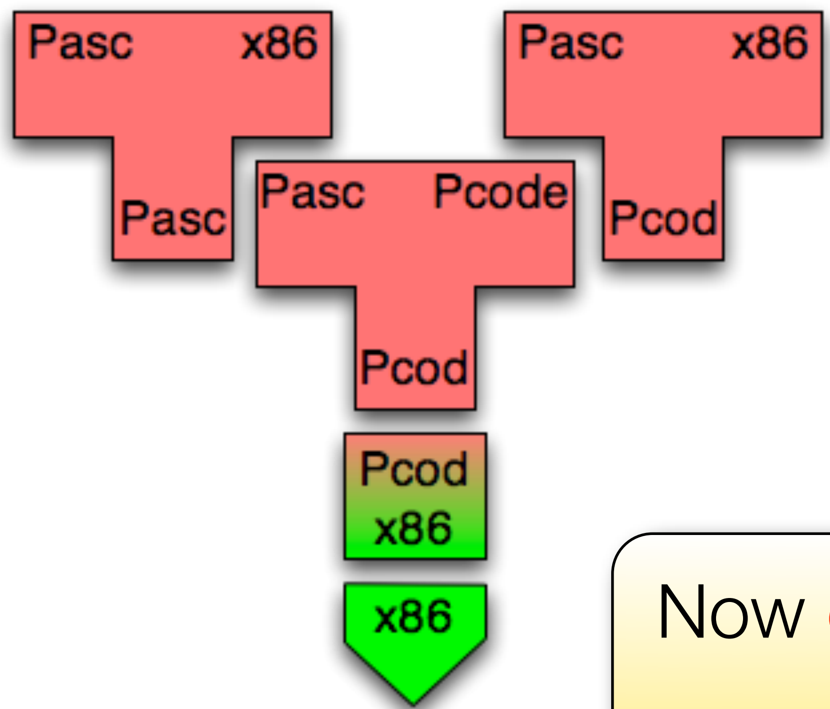


Bootstrapping (Faster)

This is **MUCH HARDER** than creating a P-Code to Machine language interpreter.

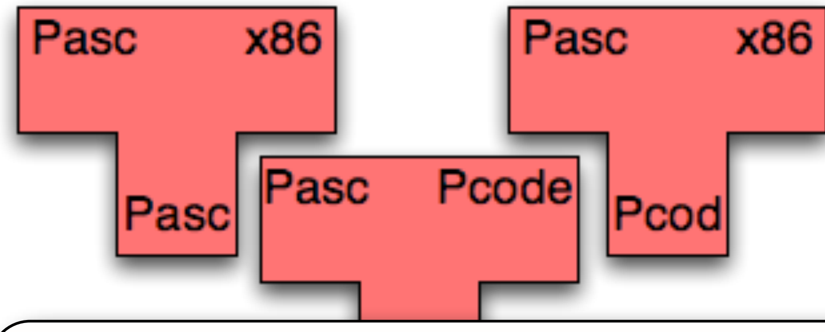


Bootstrapping (Faster)

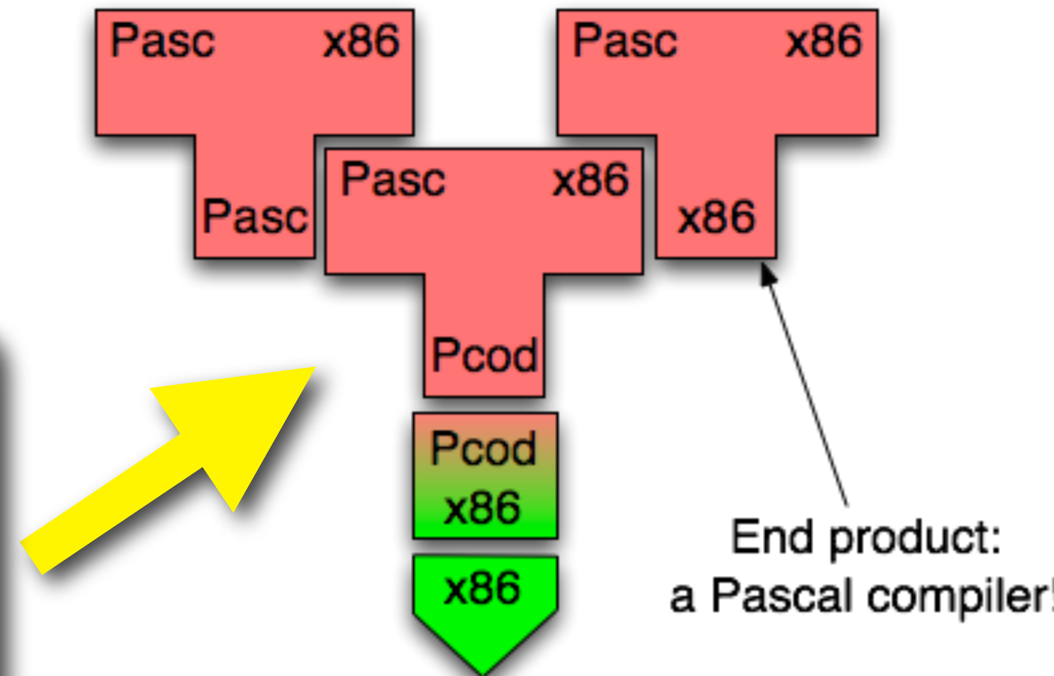


Now **construct** the “Pascal to machine lang. compiler, in **P-Code**.”

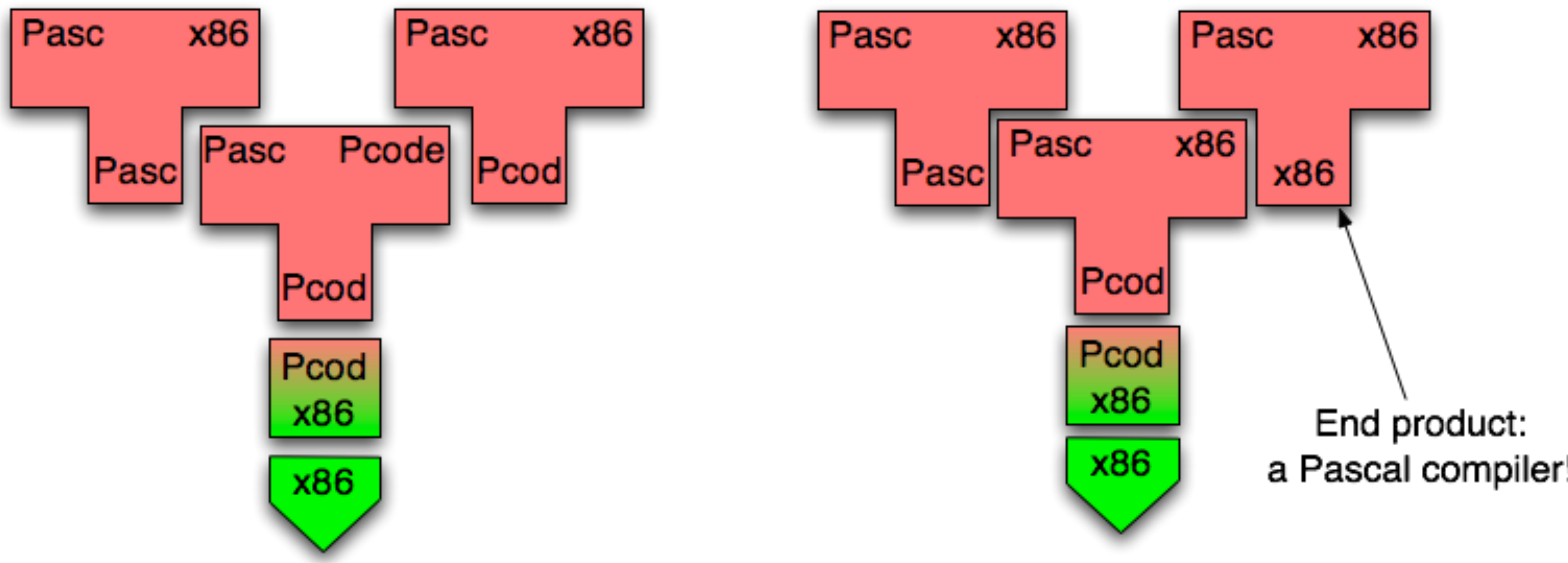
Bootstrapping (Faster)



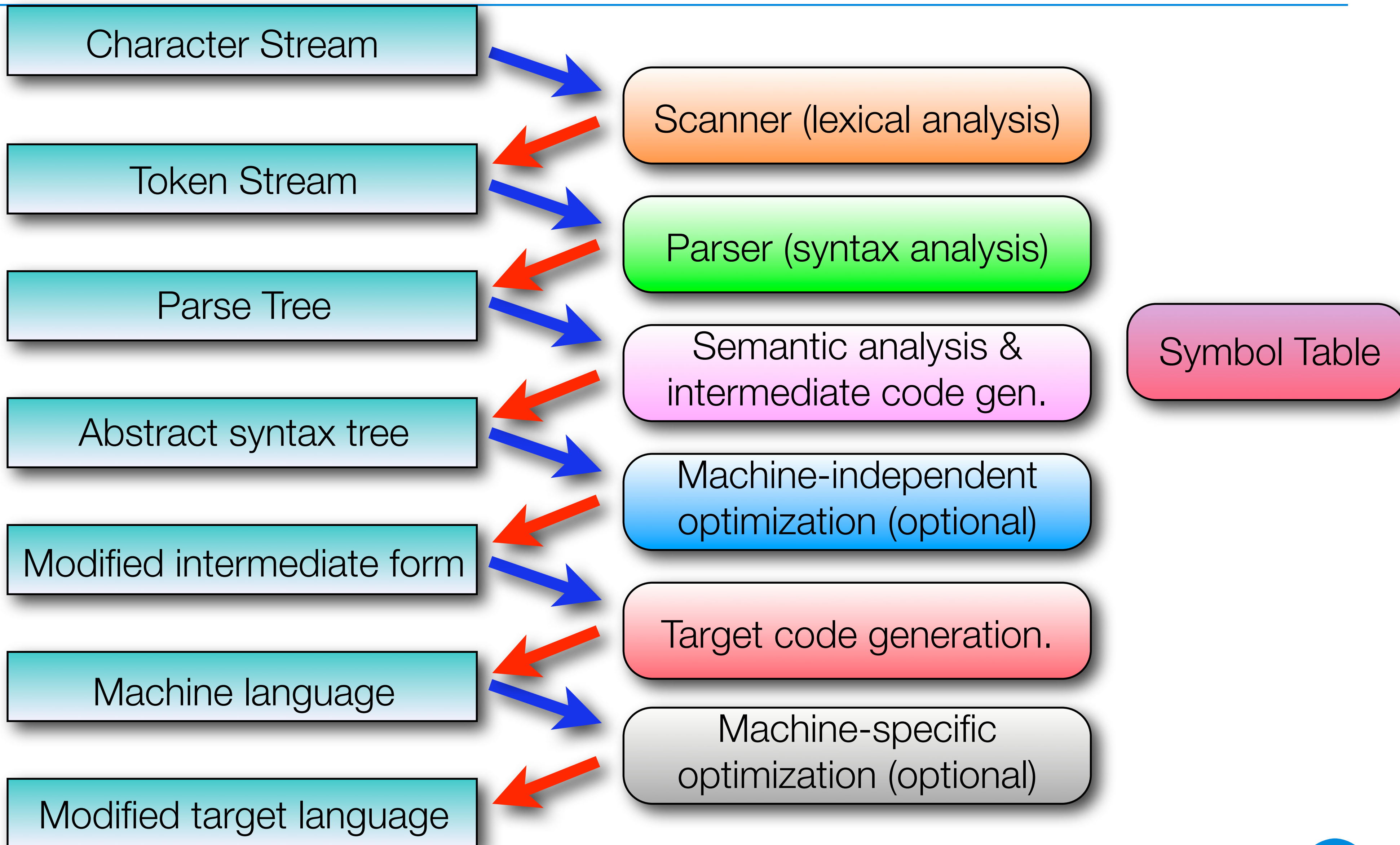
Then run the "Pascal to machine lang. compiler, in **Pascal**" through the "**P-Code**" version to produce the "**Machine lang.**" version.



Bootstrapping (Faster)



Compiling


Character Stream
Scanner (lexical analysis)
Token Stream
Parser (syntax analysis)
Parse Tree
Semantic analysis & intermediate code gen.
Symbol Table
Abstract syntax tree
Machine-independent optimization (optional)
Modified intermediate form
Target code generation.
Machine language
Machine-specific optimization (optional)
Modified target language

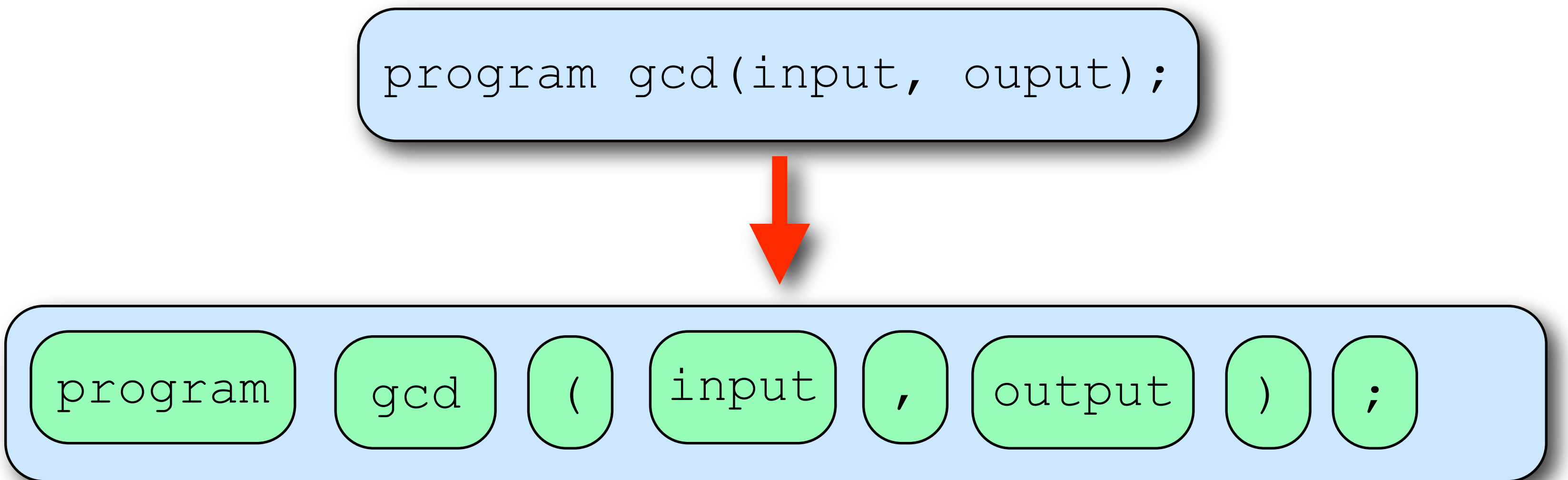
Example Program GCD

```
program gcd(input, output);
var i, j: integer;
begin
  read(i,j); // get i & j from read
  while i<>j do
    if i>j then i := i-j
    else j := j-1;
  writeln(i)
end.
```

Lexical Analysis

Scanner (lexical analysis)

- Recognize structures without regard to meaning and groups them into **tokens**.

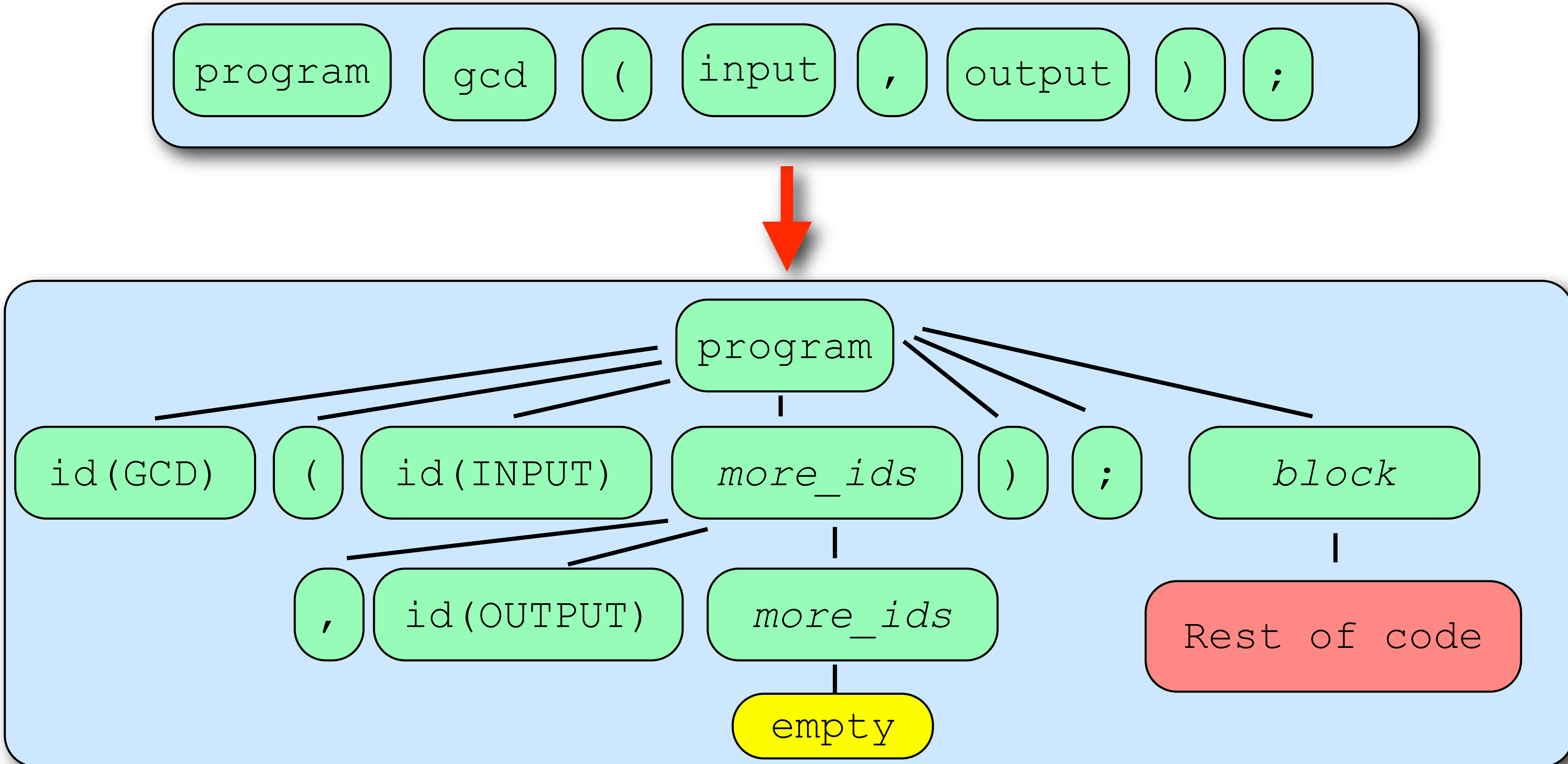


- The purpose of the scanner is to simplify the parser by **reducing the size of the input**.

Syntax Analysis

Parser (syntax analysis)

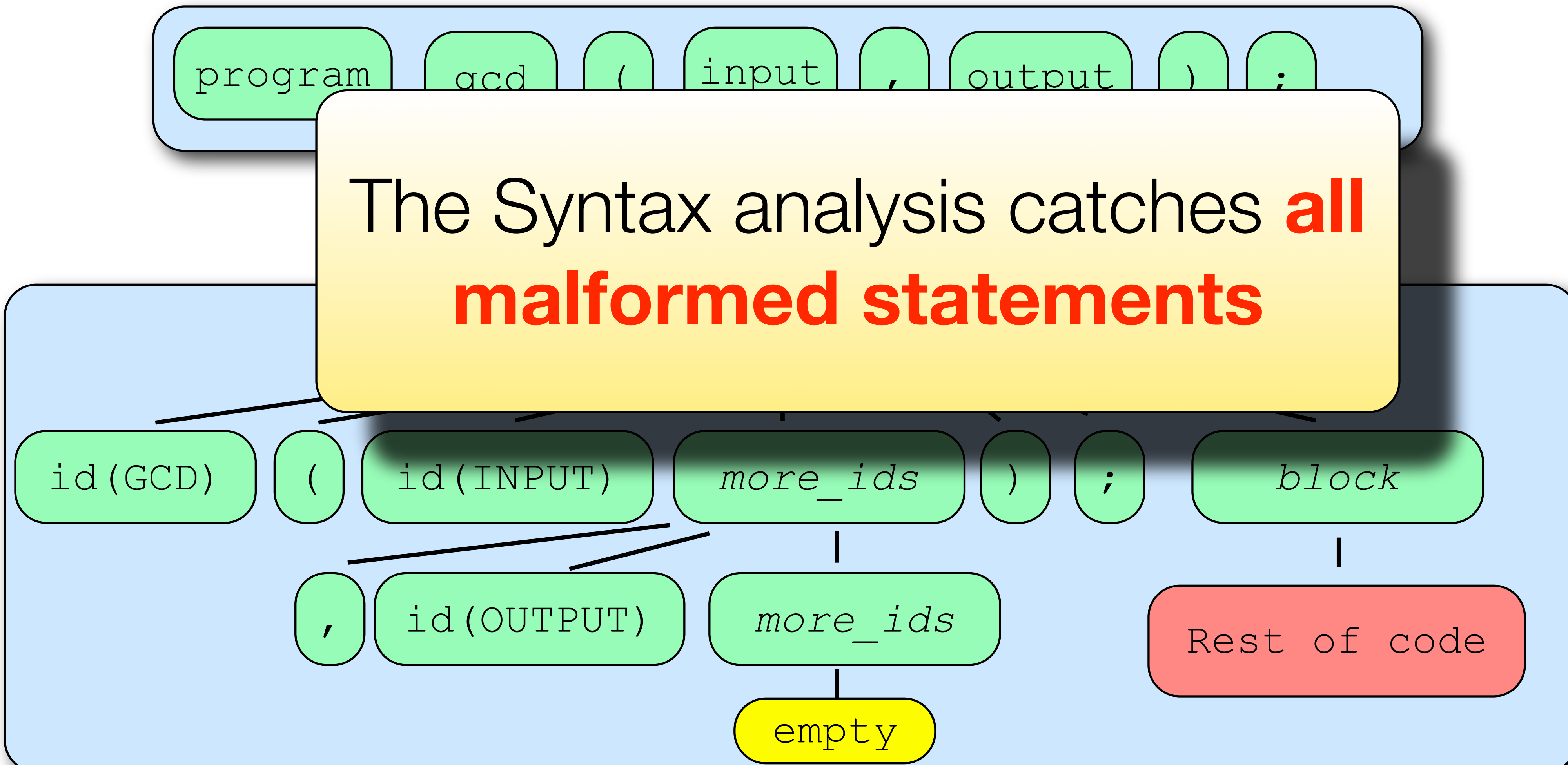
- Parsing organizes the tokens into **a context-free grammar** (i.e., **syntax**).



Syntax Analysis

Parser (syntax analysis)

- Parsing organizes the tokens into **a context-free grammar** (i.e., **syntax**).



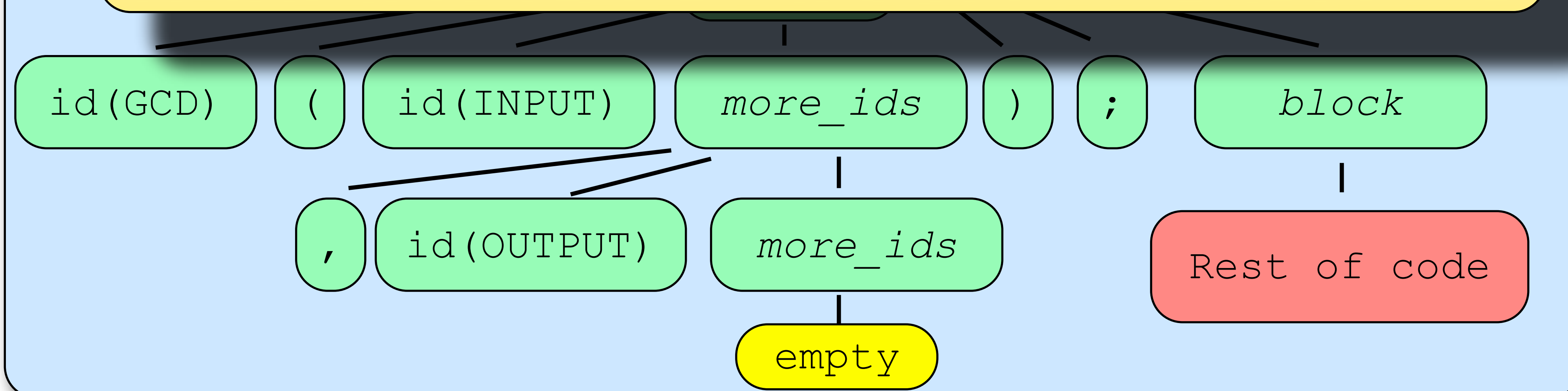
The Syntax analysis catches **all malformed statements**

Syntax Analysis

Parser (syntax analysis)

- Parsing organizes the tokens into **a context-free g**

The parse tree is sometimes called a **concrete syntax tree** because it contains how **all tokens are derived**...

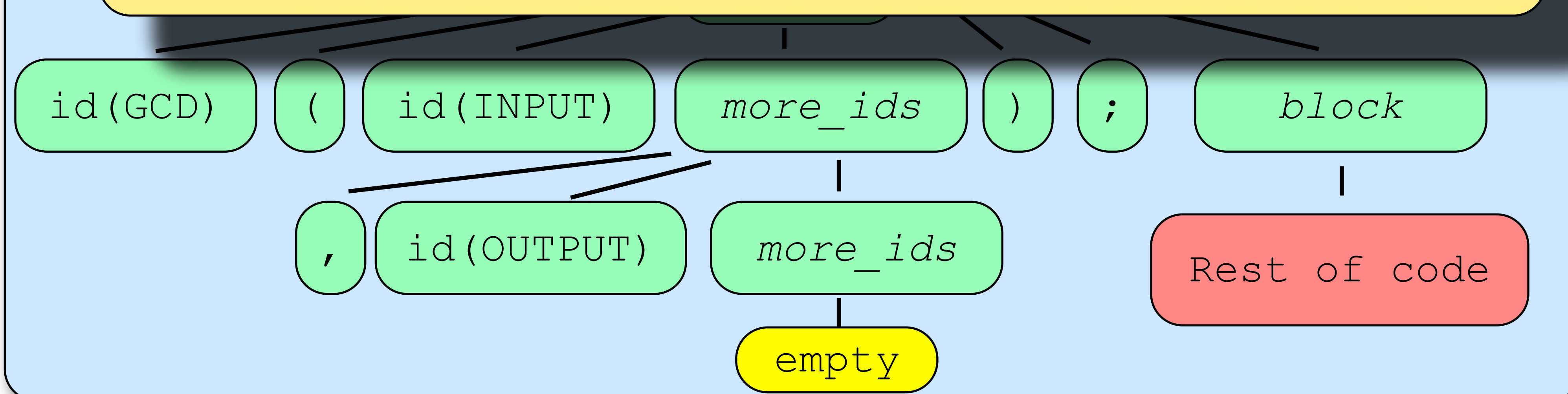


Syntax Analysis

Parser (syntax analysis)

- Parsing organizes the tokens into **a context-free**

...however, much of this information is **extraneous** for the "meaning" of the code(e.g., the only purpose of "`;`" is to **end a statement**).



Semantic Analysis

Semantic analysis & intermediate code gen.

- Semantic analysis discovers the **meaning of a program**. by creating an **abstract syntax tree** that removes "extraneous" tokens.
- To do this, the analyzer builds & maintains a **symbol table** to map identifiers to information known about it. (i.e., scope, internal structure, etc...)
- By using the symbol table, the semantic analyzer can **catch problems not caught by the parser**. For example,
  - Identifiers are declared before used
  - subroutine calls provide correct number and type of arguments.

Semantic Analysis

Semantic analysis & intermediate code gen.

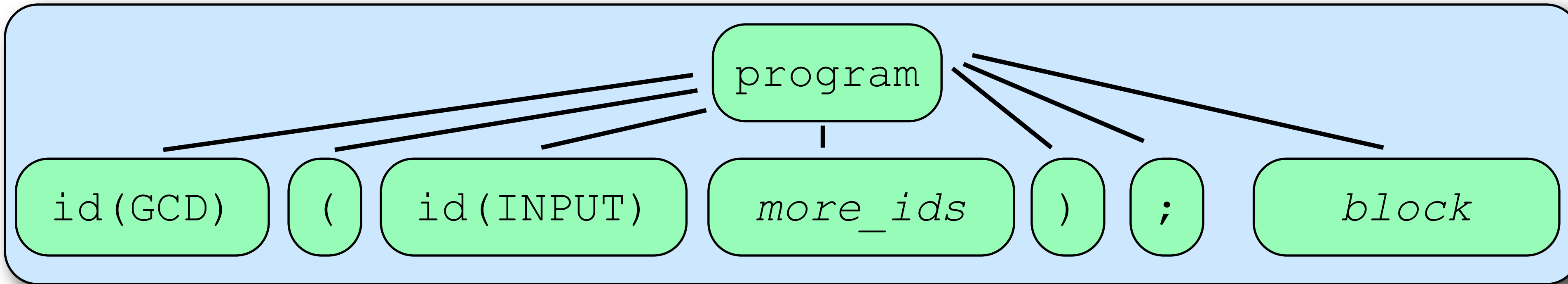
- Not all semantic rules can be checked at compile time.
  - Those that can are called **static** semantics of the language.
  - Those that cannot are called **dynamic** semantics of the language. For example,
    - Arithmetic operations **do not overflow**.
    - Array subscripts expressions **lie within the bounds of the array**.

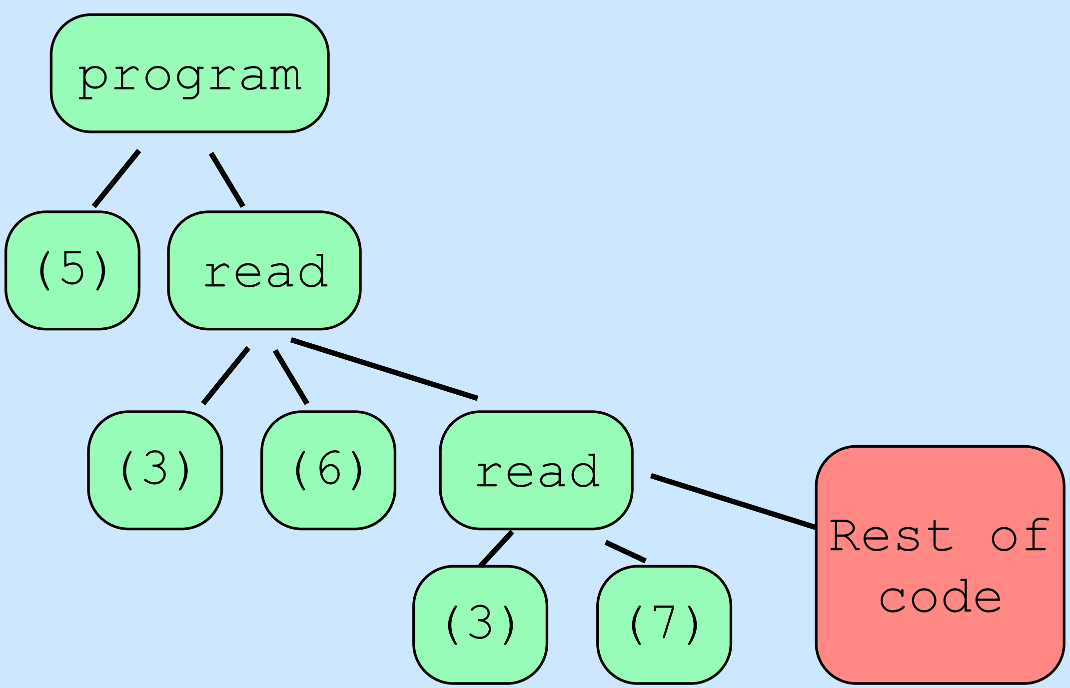
Example Program GCD

```
program gcd(input, output);
var i, j: integer;
begin
  read(i,j); // get i & j from read
  while i<>j do
    if i>j then i := i-j
    else j := j-1;
  writeln(i)
end.
```

Semantic Analysis

Semantic analysis & intermediate code gen.





Index	Symbol	Type
1	INTEGER	type
2	TEXTFILE	type
3	INPUT	2
4	OUTPUT	2
5	GCD	program
6	I	1
7	J	1

Target code generation

Target code generation.

- Code generation takes the abstract syntax tree and the symbol table to produce machine readable code.
- **Simple** code follows directly from the abstract syntax tree and symbol table.

Optimization

Machine-independent optimization (optional)

Machine-specific optimization (optional)

- The process so far will produce **correct code**, but it **may not be fast**.
- Optimization will adjust the code to improve performance.
  - A possible machine-indp. optimization would be to keep the variables `i` and `j` in **registers throughout the main loop**.
  - A possible machine-spec. optimization would be to assign the variables `i` and `j` to **specific registers**.