Fundamentals

Mitchell Chapter 4

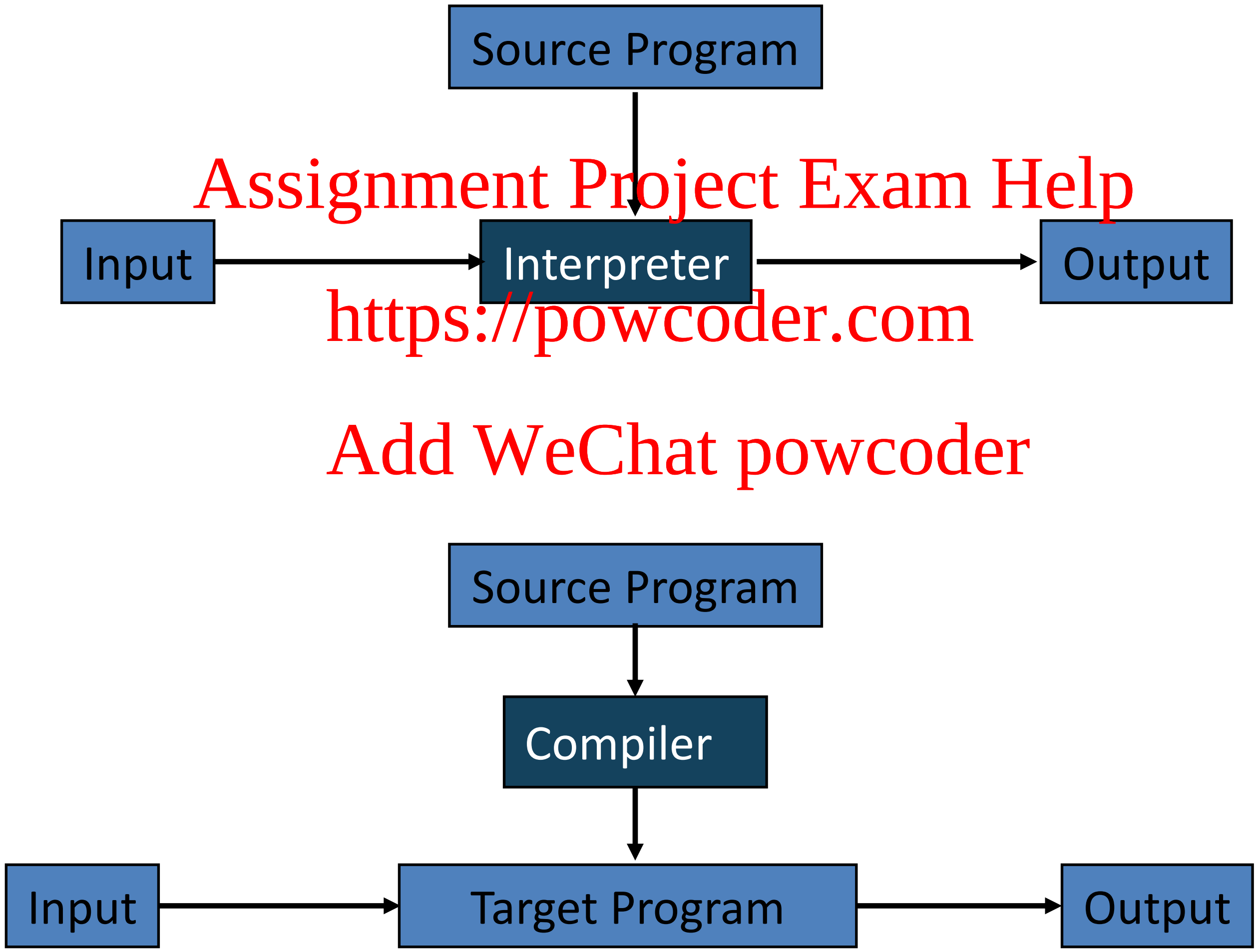
Author John Mitchell, updated by Amy Felty

Syntax and Semantics of Programs

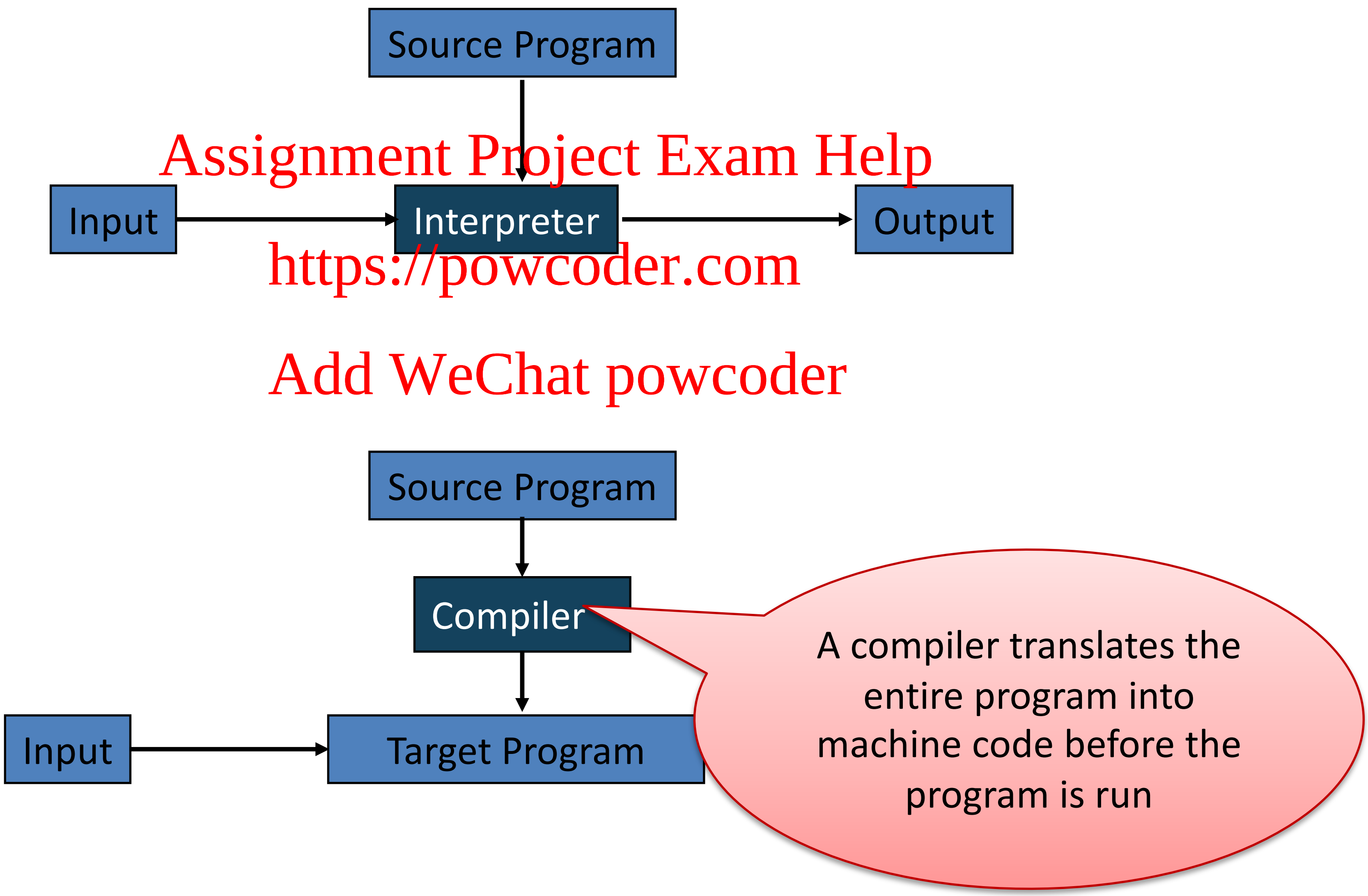
"…theoretical frameworks have had an impact on the design of programming languages and can be used to identify problem areas in programming languages."

- Syntax
  - The symbols used to write a program
- Semantics
  - The actions that occur when a program is executed
- Programming language implementation
  - Syntax → Semantics
  - Transform program syntax into machine instructions that can be executed to cause the correct sequence of actions to occur

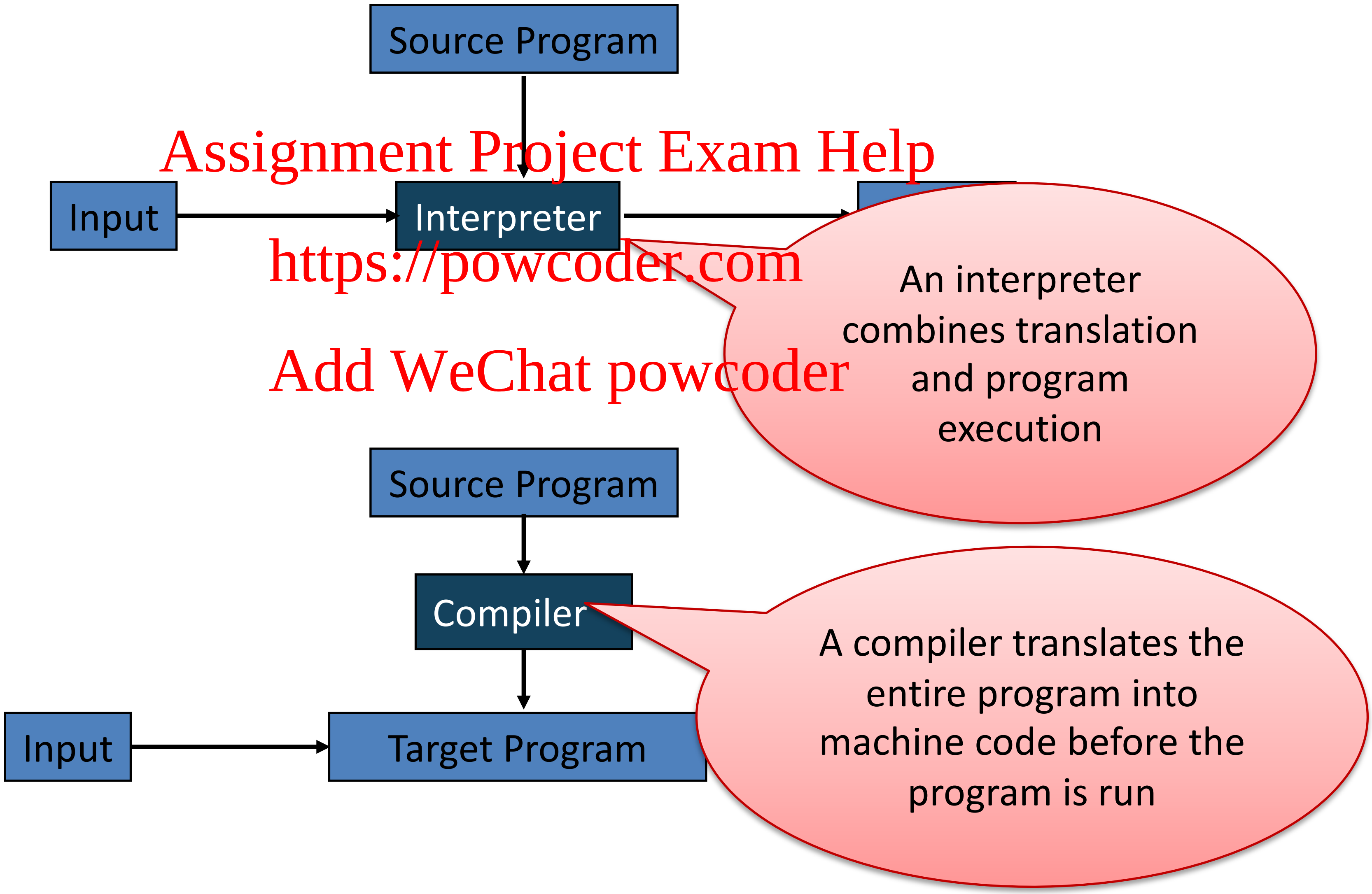
Interpreters vs. Compilers



Interpreters vs. Compilers

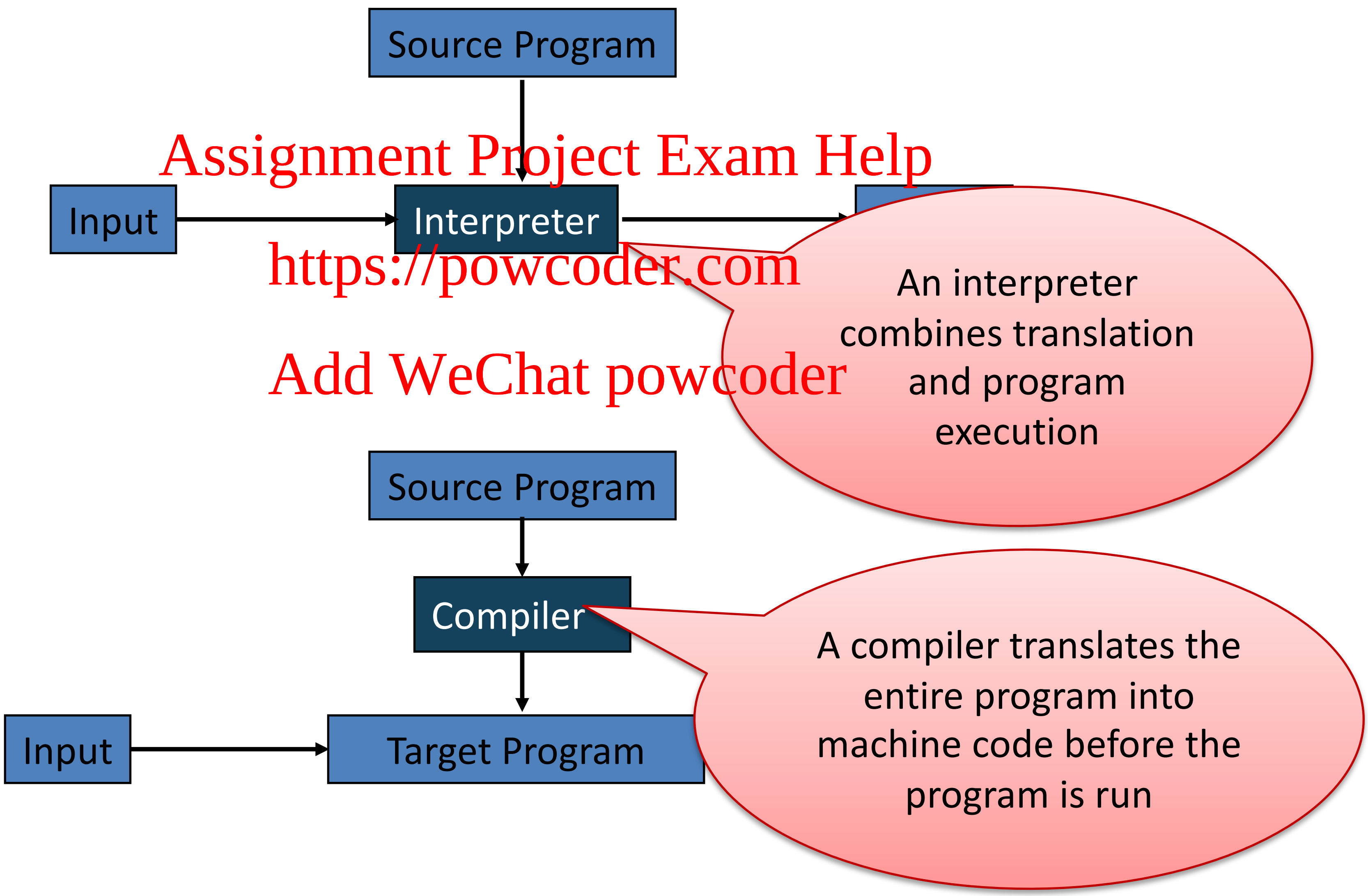


Interpreters vs. Compilers

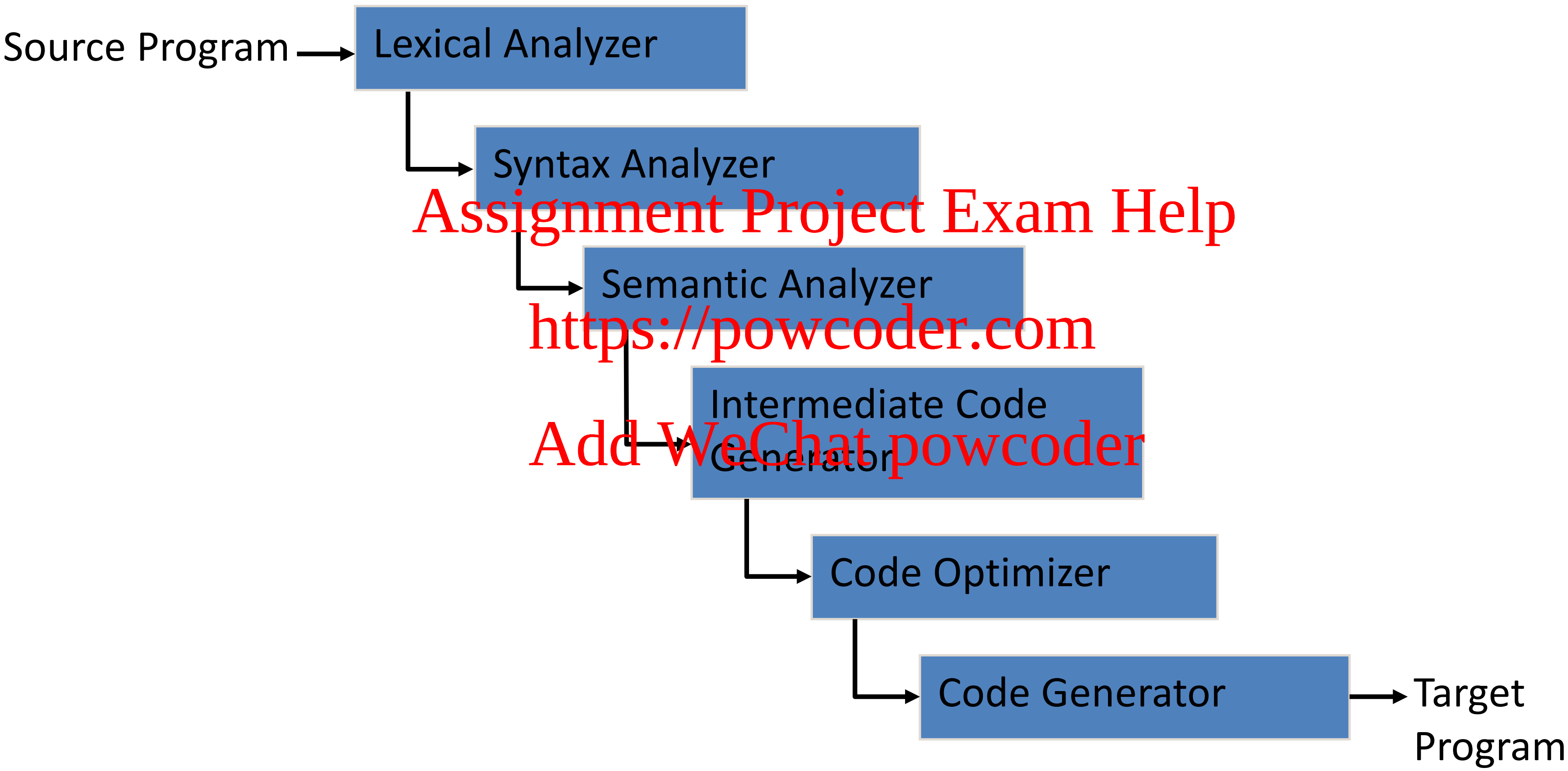


Interpreters vs. Compilers

Studying compilers makes it easier to separate the main issues and discuss them in a given order.



A Typical Compiler



Compiler Phases

- Lexical Analysis
  - Input symbols are scanned from left to right and grouped into meaningful units called *tokens*.
  - Distinguishes numbers, identifiers, symbols and keywords.
  - Example: temp := x+1

    Tokens are: temp, :=, x, +, 1

- Syntax Analysis

  - Parsing: tokens are grouped into syntactic units such as expressions, statements, and declarations that must conform to the grammatical rules of the programming language.

  - If the program does not meet the syntactic requirement to be a well-formed program, an error message is reported, and the compiler terminates.
  - The result is a parse tree.
  - To be discussed in more detail.

Compiler Phases

- Semantic Analysis
  - Context information is used to augment the parse tree, i.e., type information (from type inference)
  - Note the difference between semantic analysis and program semantics (i.e. program meaning)
- Intermediate Code Generation
  - It is difficult to generate efficient code in one phase.
  - It is important to use an intermediate representation that is easy to produce and easy to translate into the target language.
- Code Optimization
  - Different techniques are applied over and over to the intermediate representation. (See next page.)
- Code Generation
  - Converts the intermediate code into a target machine code.
  - Involves choosing memory locations and registers for variables.
  - Efficiency is important.

Some Standard Code Optimizations

- Common Subexpression Elimination
  - If a program calculates the same value more than once, then calculate only once and store for later use.
- Copy Propagation
  - If a program contains an assignment x=y then it may be possible to change later statements to refer to y instead of to x and remove the assignment.

- Dead-Code Elimination
  - Eliminate sequences of code that can never be reached.
- Loop Optimizations
  - Move expressions that occur inside a loop to outside the loop if they don’t change value.
- In-lining Function Calls
  - Substitute function calls with the body of the function when possible. This often allows further optimizations to be performed by removing jumps.

Syntax: Grammars and Parse Trees

- Grammar

  e ::= n | e+e | e–e

  n ::= d | nd

  d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- Expressions in language generated by *derivations*, e.g.,

e → e–e

→ e–e+e → n–n+n → nd–d+d → dd–d+d

→ … → 27 – 4 + 3

Grammar defines a language

Expressions in language derived by sequence of productions

Syntax: Grammars and Parse Trees

- Grammar

  e ::= n | e+e | e–e

  n ::= d | nd

  d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- A Grammar includes:
  - A *start symbol* (e in this case)
  - A set of *nonterminals*
  - A set of *terminals* (which appear in the expressions of the language generated by the grammar)

- In this example:
  - Nonterminals: e, n, d
  - Terminals: 0,…9,+, –
- Examples:
  - 0, 1+3+5, 2+4-6-8

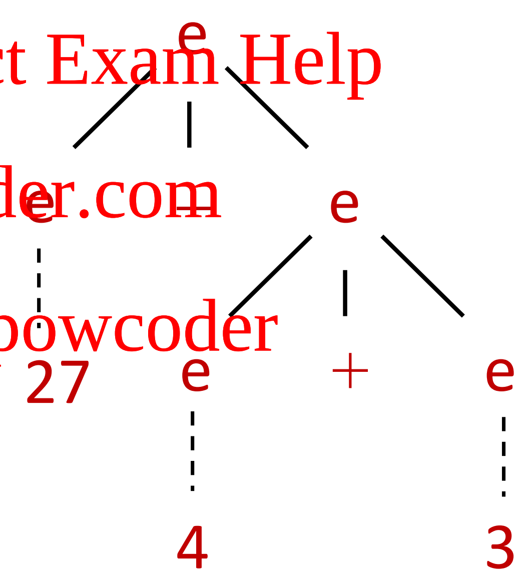
Nonterminals keep track as a valid expression is being formed. They must eventually be replaced.

Parse Trees (Derivation Trees)

- Derivation represented by tree

  $e \rightarrow e{-}e \rightarrow e{-}e{+}e \rightarrow n{-}n{+}n \rightarrow nd{-}d{+}d \rightarrow dd{-}d{+}d$

  $\rightarrow \ldots \rightarrow 27 - 4 + 3$



Tree shows parenthesization of expression.
A grammar is *ambiguous* if some expression
has more than one parse tree.

Parse Trees (Derivation Trees)

- Exercise: draw 2 parse trees for 10—15 + 12

- Grammar

  s ::= v:=e | s;s | if b then s | if b then s else s

  v ::= x | y | z

  e ::= v | 0 | 1 | 2 | 3 | 4

  b ::= e=e

- Exercise: draw 2 parse trees for

  if b1 then if b2 then s1 else s2

  What happens when b1=true and b2=false?

Mitchell, pp. 54-55

Parsing

- Parsing
  - Given a language *L* defined by a grammar *G*, and a string of symbols *s*, an algorithm that decides whether *s* is in *L*, and constructs a parse tree if it is, is called a *parsing algorithm* for *G*.
- Ambiguity

  - Expression 27 – 4 + 3 can be parsed two ways
  - Problem: 27 – (4 + 3) ≠ (27 – 4) + 3

- Ways to resolve ambiguity

  - Precedence
    - By convention * has higher *precedence* than + or —
    - For example, parse 3*4 + 2 as (3*4) + 2
  - Associativity
    - Parenthesize operators of equal precedence to left (or right)
    - Parse 3 – 4 + 5 as (3 – 4) + 5