Fundamentals

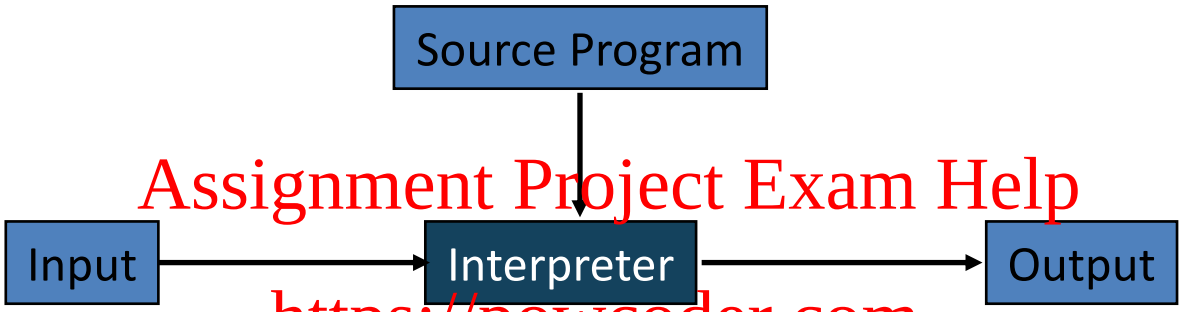
Mitchell Chapter 4

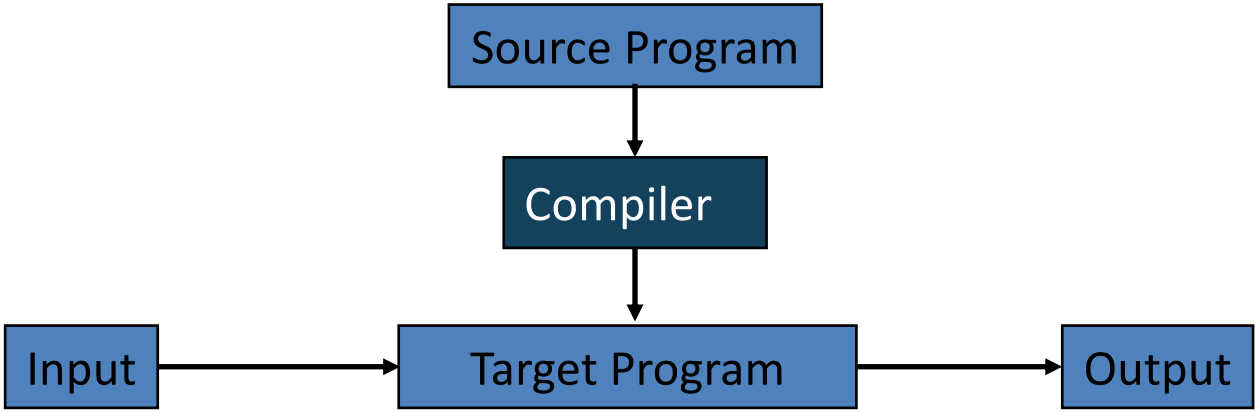
Syntax and Semantics of Programs

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

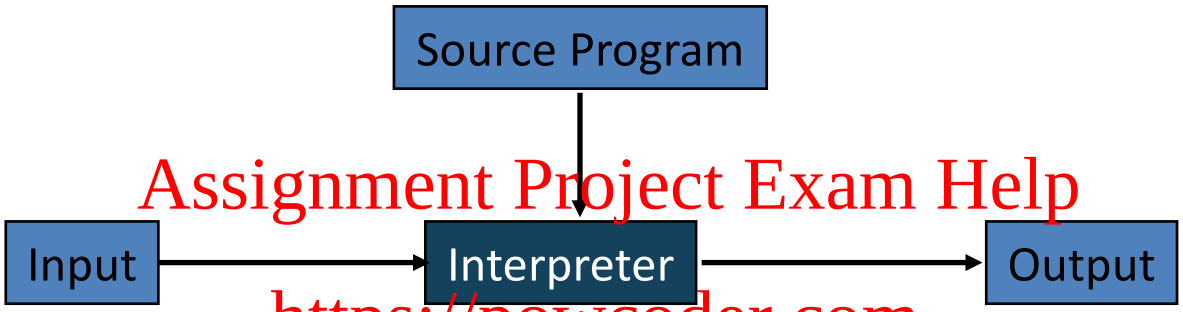
- Syntax
  - The symbols used to write a program
- Semantics
  - The actions that occur when a program is executed
- Programming language implementation
  - Syntax → Semantics
  - Transform program syntax into machine instructions that can be executed to cause the correct sequence of actions to occur

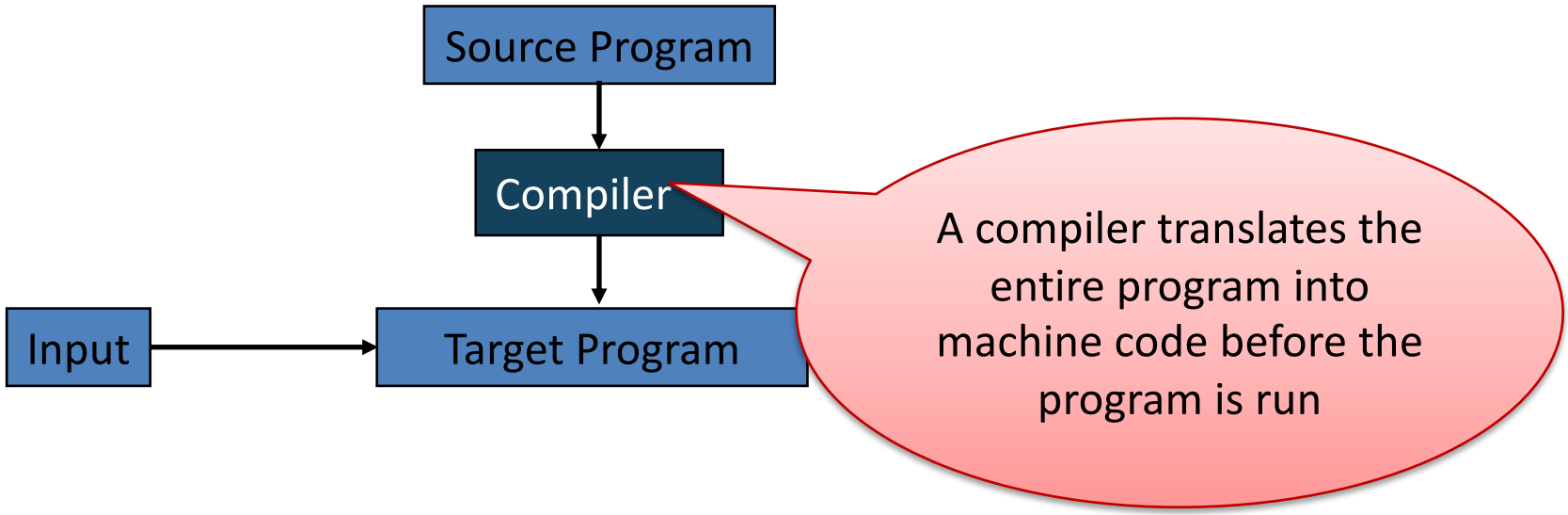
Interpreters vs. Compilers



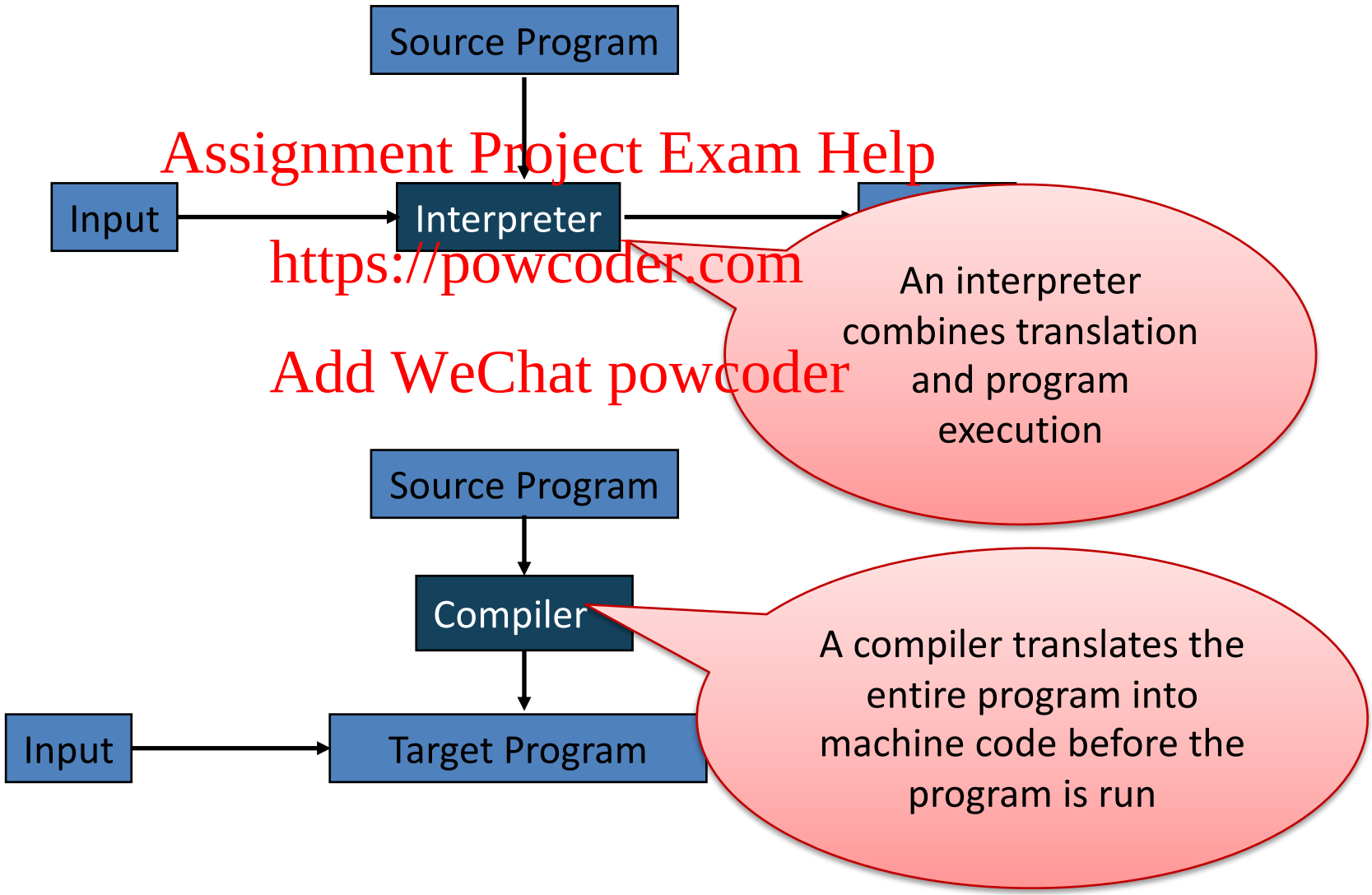


Interpreters vs. Compilers



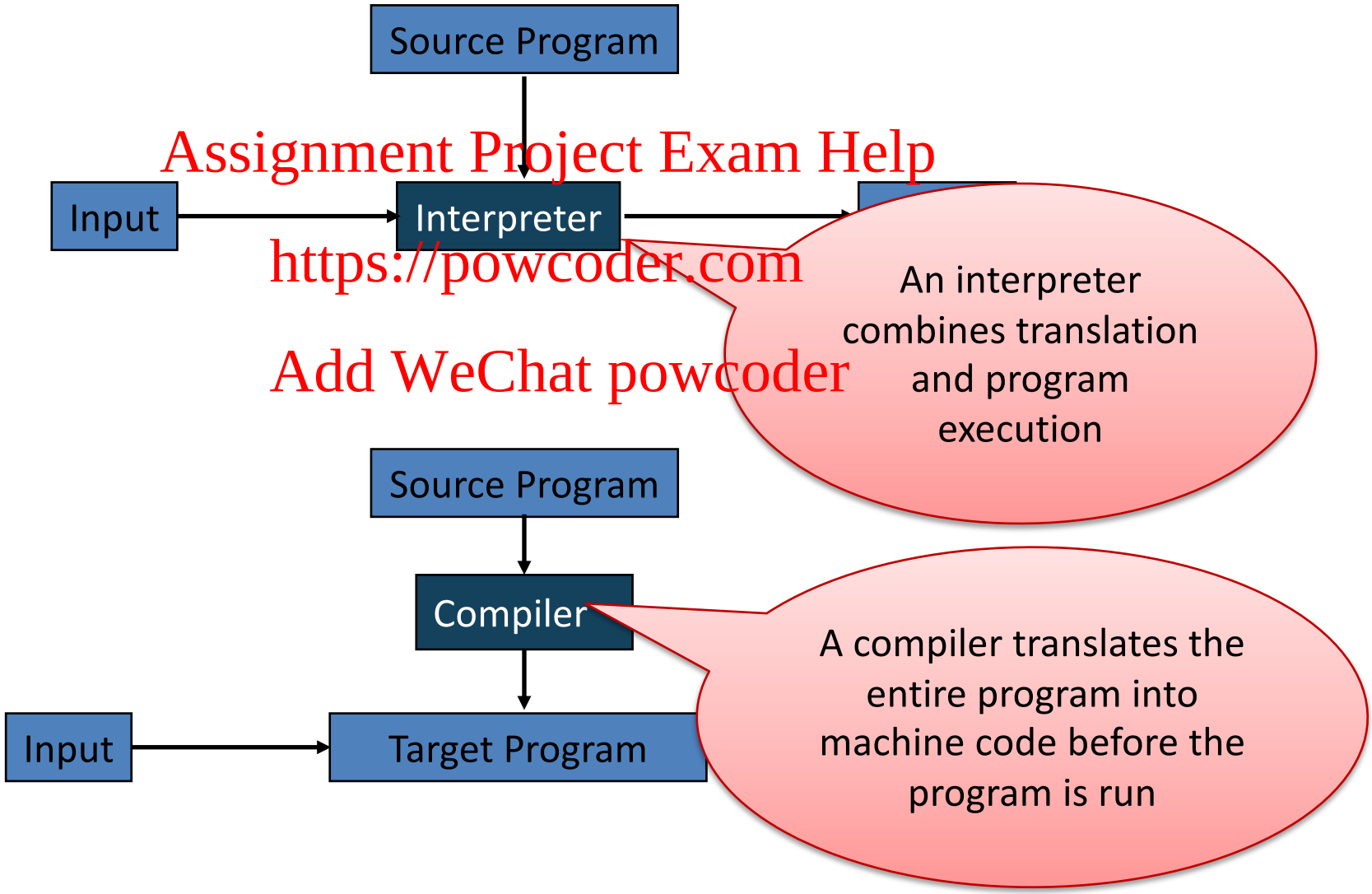


Interpreters vs. Compilers

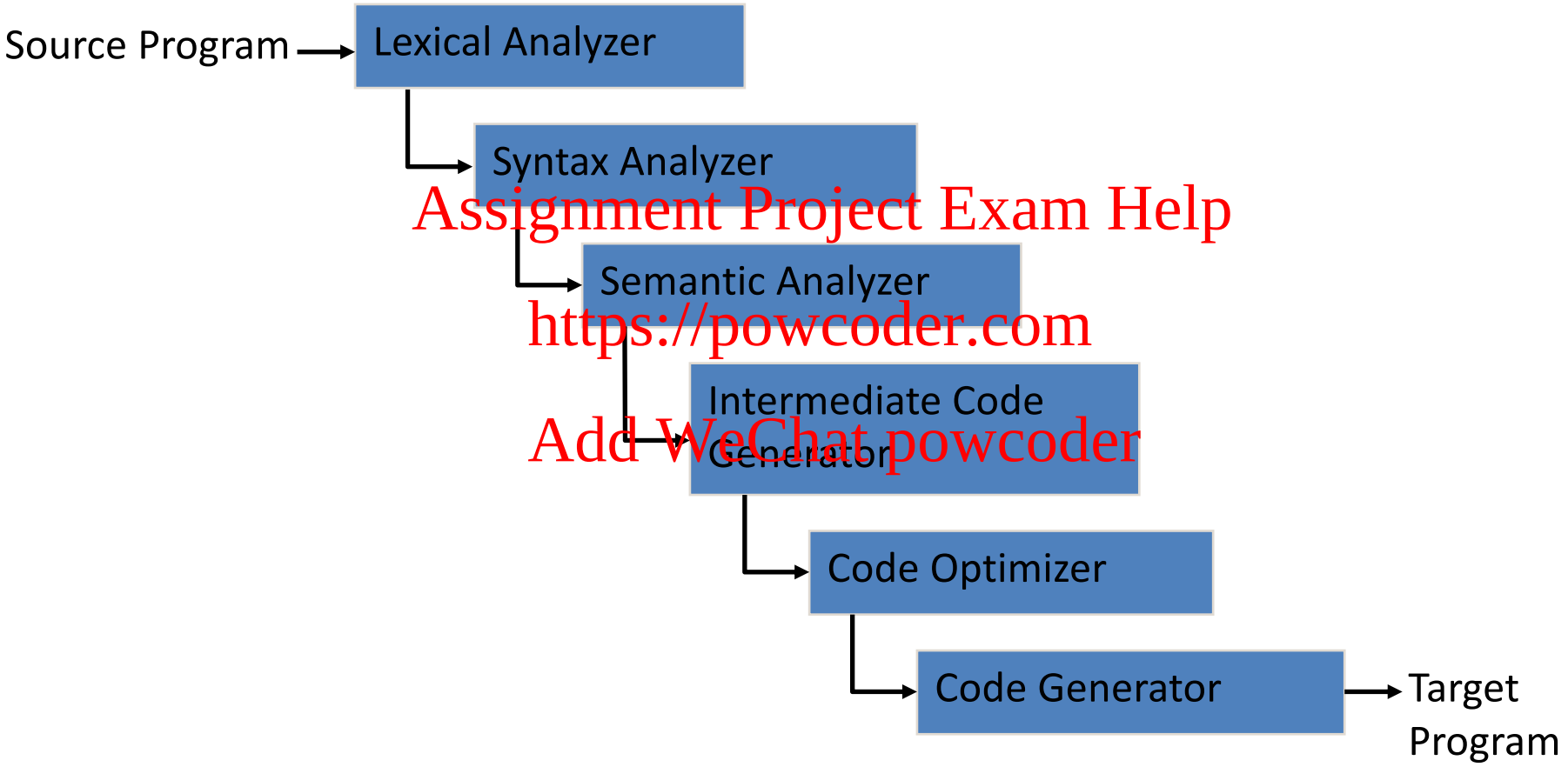


Interpreters vs. Compilers

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

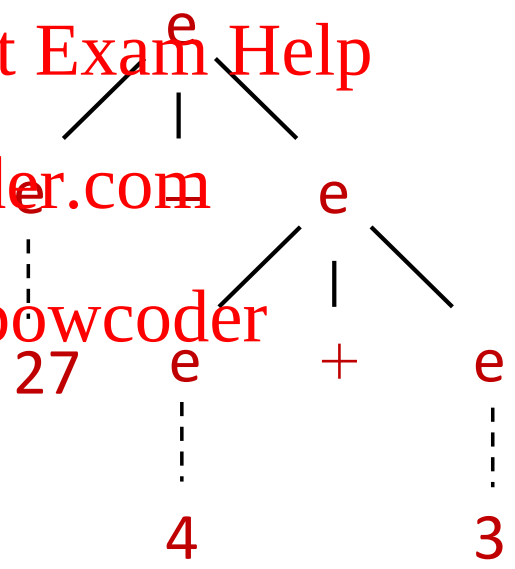
Nonterminals keep track as a valid expression is being formed. They must eventually be replaced.

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: $27 - (4 + 3) \neq (27 - 4) + 3$
- 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

Theoretical Foundations

- There are many foundational systems.
  - Computability Theory
  - Program Logics
  - Lambda Calculus
  - Denotational Semantics
  - Operational Semantics
  - Axiomatic Semantics
  - Type Theory
- We will consider two of these methods.
  - Lambda calculus (syntax, operational semantics)
  - Axiomatic semantics
- We will also discuss (again):
  - Functional vs imperative programming

Lambda Calculus

- Lambda calculus
  - a mathematical system that illustrates some important programming language concepts in a simple, pure form.
- Formal system with three parts
  - Notation for function expressions
  - Proof system for equations
  - Calculation rules called *reduction* or *evaluation*
- Additional topics in lambda calculus
  - Mathematical semantics (=model theory)
  - Type systems

We will look at syntax, equations, and reduction

History

- Original intention
  - Formal theory of substitution (for FOL, etc.)
- More successful for computable functions
  - Substitution --> symbolic computation
  - Church/Turing thesis
- Influenced design of Lisp, ML, other languages
- Important part of CS history and foundations

Why study this now?

- Lambda
  - You have seen "lambda" appear in many languages: Python, Scheme, and now OCaml
- Basic Syntactic Notions
  - Free and bound variables
  - Functions
  - Declarations
- Calculation Rule
  - Symbolic evaluation useful for discussing programs
  - Used in optimization (in-lining), macro expansion
    - Correct macro processing requires variable renaming
  - Illustrates some ideas about scope of binding
    - Lisp originally departed from standard lambda calculus, returned to the fold through Scheme, Common Lisp

Expressions and Functions

- Grammar

  e ::= x | e e | λx.e

  - x represents an infinite set of variables $\{x, y, z, x_1, x_2, x_3, \ldots\}$

  - Lambda abstraction and application are the main constructs

  - Programming language = applied lambda-calculus = pure lambda-calculus + additional data types

  - Here we use numbers.
  - In λx.e, e is called the scope of x.

Expressions and Functions

- Grammar

  e ::= x | e e | λx.e

- Expressions

  x + y x + 2*y + z

- Functions

  λx. (x+y) λz. (x + 2*y + z)

- Application

  (λx. (x+y)) 3 = 3 + y

  (λz. (x + 2*y + z)) 5 = x + 2*y + 5

Parsing: λx. f (f x) = λx.(f (f (x)))

Free and Bound Variables

- Bound variable is "placeholder"
  - Variable x is bound in $\lambda x.\ (x+y)$
  - Function $\lambda x.\ (x+y)$ is same function as $\lambda z.\ (z+y)$
- Compare

  $\int x+y\ dx = \int z+y\ dz \qquad \forall x\ P(x) = \forall z\ P(z)$

- Name of free (=unbound) variable does matter
  - Variable y is free in $\lambda x.\ (x+y)$
  - Function $\lambda x.\ (x+y)$ is *not* same as $\lambda x.\ (x+z)$

- Occurrences
  - y is *free* and *bound* in $\lambda x.\ ((\lambda y.\ y + 2)\ x) + y$

Equivalence and Substitution

- α-equivalence

$$\lambda x.\ e \ = \lambda y.[y/x]e$$

- β-equivalence

$$(\lambda x.\ e_1)\ e_2 \ = \ [e_2/x]e_1$$

- Substitution
  - $[e_2/x]e_1$ is the result of substituting e_2 for free occurrences of x in e_1.

Reduction

- Basic computation rule is β-reduction

$$(\lambda x.\, e_1)\, e_2 \;\rightarrow\; [e_2/x]e_1$$

where substitution involves renaming as needed

- Reduction:

  - Apply basic computation rule to any subexpression
  - Repeat

- Confluence:

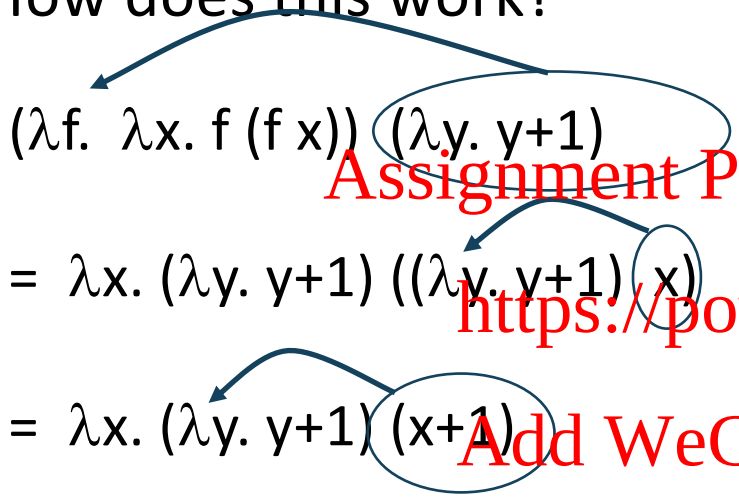
  - Final result (if there is one) is uniquely determined

Higher-Order Functions

- Given function f, return function f ∘ f

  λf. λx. f (f x)

- How does this work?



(λf. λx. f (f x)) (λy. y+1)

= λx. (λy. y+1) ((λy. y+1) x)

= λx. (λy. y+1) (x+1)

= λx. (x+1)+1

Same result if step 2 is delayed until after step 3.

Same procedure, Lisp syntax

- Given function f, return function f^2

  (lambda (f) (lambda (x) (f (f x))))

- How does this work?

  ((lambda (f) (lambda (x) (f (f x)))) (lambda (y) (+ y 1))

  = (lambda (x) ((lambda (y) (+ y 1))
  ((lambda (y) (+ y 1)) x)))

  = (lambda (x) ((lambda (y) (+ y 1)) (+ x 1))))

  = (lambda (x) (+ (+ x 1) 1))

Declarations as "Syntactic Sugar"


```
function f(x)
    return x+2
end;
f(5);
```


$(\lambda f.\ f(5))\ (\lambda x.\ x+2)$

block body — declared function

let x = e_1 in e_2 = $(\lambda x.\ e_2)\ e_1$

Rename Bound Variables

- Function application

  $(\lambda f.\ \lambda x.\ f\ (f\ x))\ \ (\lambda y.\ y+x)$

  apply twice add x to argument

- Substitute "blindly"

  $\lambda x.\ [(\lambda y.\ y+x)\ ((\lambda y.\ y+x)\ x)]\ =\ \lambda x.\ x+x+x$

- Rename bound variables

  $(\lambda f.\ \lambda z.\ f\ (f\ z))\ \ (\lambda y.\ y+x)$

  $=\ \lambda z.\ [(\lambda y.\ y+x)\ ((\lambda y.\ y+x)\ z))]\ =\ \lambda z.\ z+x+x$

  Easy rule: always rename variables to be distinct

Substitution: A Precise Definition

Substituting e for x in e' is written [e/x]e' and is defined inductively on the structure of e' as follows:

- [e/x]x = e
- [e/x]y = y, where y is any variable different from x
- [e/x](e_1 e_2) = ([e/x]e_1) ([e/x]e_2)
- [e/x] (λx.e_1) = λx.e_1
- [e/x] (λy.e_1) = λy.([e/x]e_1), where y is not free in e

- Because we are free to rename the bound variable y in λy.e', the final clause always makes sense.
- With this precise definition, we now have a precise definition of α-equivalence and β-reduction.

Main Points about Lambda Calculus

- λ captures "essence" of variable binding
  - Function parameters
  - Declarations
  - Bound variables can be renamed
- Succinct function expressions

- Simple symbolic evaluator via substitution

- Can be extended with
  - Types

  - Various functions
  - Stores and side-effects

(But we didn't cover these)

Summary of Lambda Calculus

- Lambda calculus is a mathematical system with some syntactic and computational properties of a programming language.
- There is a general notation for functions that includes a way of treating an expression as a function of some variable that it contains.

λx.e where x is a variable and e is the expression that contains x

Summary of Lambda Calculus

- Lambda calculus is a mathematical system with some syntactic and computational properties of a programming language.
- There is a general notation for functions that includes a way of treating an expression as a function of some variable that it contains.
- There is an equational proof system that leads to calculation rules, which can be viewed as a simple form of symbolic evaluation.
- In programming language terminology, these calculation rules are a form of macro expansion (with renaming of bound variables) or function in-lining.
- Because of the relation to in-lining, some common compiler optimizations may be defined and proved correct by use of lambda calculus.

Functional vs. Imperative Programs

- Imperative vs. Declarative Sentences in English
  - Imperative example (command): Pick up that fish.
  - Declarative example (fact): Claude likes bananas.
- Imperative vs. Declarative in Programming
  - Imperative: changing a value
  - Declarative: declaring a new value


```
{ int x = 1;                    /* declares new x */
  x = x + 1;                    /* assignment to existing x */
  { int y = x+1;                /* declares new y */
    { int x = y+1;              /* declares new x */
```


  - Only the second line is imperative.
  - Note that the last line declares a new variable with the same name.
  - Recall that let statements in OCaml are declarations.

Renaming "Bound" Variables in Programs


```
{ int x = 1;              /* declares new x */
  x = x + 1;              /* assignment to existing x */
  { int y = x+1;          /* declares new y */
    { int z = y+1;        /* declares new z */
```


- The simplest way to understand the distinction between declaring a new variable and changing the value of an old one is by variable renaming.

- Lambda calculus: bound variables can be renamed without changing the meaning of an expression or program.

- If there were any additional occurrences of x in the inner block, we would rename them to z also.

The Declarative Language Test

- Pure functional languages (no side effects) pass the Declarative Language Test:
  - Within the scope of specific declarations of $x_1,\ldots,x_n$, all occurrences of an expression e containing only variables $x_1,\ldots,x_n$ have the same value.

- As a consequence, pure functional languages have a useful optimization property:

  - If expression e occurs several places within a specific scope, this expression needs to be evaluated only once.

  - For example, suppose a program in Pure Lisp (or OCaml without references) contains (cons a b), or equivalently (a::b). An optimizing compiler could compute this expression once and use the same value in both places.
  - This kind of optimization saves time and space.

Parallelism: A Trade-off

- Given a function call `f(e1,…,en)` where the arguments are expressions that may need to be evaluated:
  - Functional programming: We can evaluate `f(e1,…,en)` by evaluating `e1,…,en` in parallel because values of these expressions are independent.
  - Imperative programming: For an expression such as `f(g(x),h(x))`, the function `g` might change the value of `x`. Hence the arguments of functions must be evaluated in a fixed sequential order, which restricts concurrency.
- But there can be too much parallelism in a large program.
  - E.g., full parallel evaluation of `if e1 then e2 else e3` will involve evaluating all three expressions. One will be irrelevant. This can greatly detract from efficiency.

Summary

- Parsing
  - The “real” program is the disambiguated parse tree
- Lambda Calculus
  - Notation for functions, free and bound variables
  - Calculate using substitution, rename to avoid capture
- Pure Functional Programs
  - May be easier to reason about
  - Parallelism: easy to find, too much of a good thing
- Semantics of Imperative Programs (Axiomatic Semantics)
  - We will come back to this later.

Semantics of Imperative Programs Preview

- Syntax

  P ::= x:=e | if B then P else P | P;P | while B do P

- Semantics

  - C : Programs → (State → State)

  - State = Variables → Values

    would be locations → values if we wanted to model aliasing