a

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | |
|  | | Fundamentals project | | | | |  | |
|  |  | | | | | | |  |
|  | | | |  |  | | | |
|  | | | | Compiler phases |  | | | |
|  | | | | Fundamentals of programming languages Fa24 - CS314 Supervised by: Dr /Osama  TA/ Ola Mahmoud Submitted by: Basant Medhat Makram  20213771 |  | | | |
|  | | |  | | |  | | |

Table of Contents

[Fundamentals project 1](#_Toc185124258)

[Compiler phases 1](#_Toc185124259)

[1st: Introduction 3](#_Toc185124263)

[2nd: Types of compilers 4](#_Toc185124264)

[3rd: project analysis 5](#_Toc185124265)

[4th: CFG Design 6](#_Toc185124266)

[5th: project implementation 6](#_Toc185124274)

[6th: run of code 16](#_Toc185124276)

[7th: conclusion 16](#_Toc185124277)

[8th: References 17](#_Toc185124278)

# 1st: Introduction

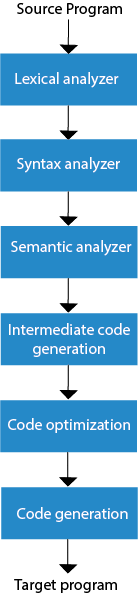
* What is Compiler?

It is special program that translates a programming language's source code into machine code, bytecode or another programming language.

* compilation process:

contains the sequence of various phases. Each phase takes source program in one representation and produces output in another representation. Each phase takes input from its previous stage.

* phases of compiler:



# 2nd: Types of compilers

#### 1. single pass compiler:

* It Reads the source code and generates machine code in a single pass.
* It is useful for processing large amounts of code quickly, especially in cases where the code is simple and straightforward. ALSO, have low overhead, which means they require less memory and processing power.
* It is useful for prototyping code before optimizing it with a more advanced compiler.

**2.Multi-pass compilers:**

* It makes multiple passes over the source code, can produce more optimized and efficient code.
* They are better at handling more complex programming languages, including forward references and recursive functions.
* They are widely used in the industry for generating optimized code for a variety of platforms, including desktops, servers, and embedded systems.

**3. Just-in-time compilers (JIT):**

* dynamically generates machine code at runtime, rather than translating the entire program before execution.
* They are commonly used in virtual machines, such as those used for Java and .NET, to improve performance by optimizing the code during execution.
* It generates a low-level intermediate code that is optimized for runtime performance then compiled into machine code on-the-fly as the program runs.
* The generated code is stored in memory and can be reused as needed.
* used in web browsers to improve the performance of JavaScript, which is an interpreted language and in virtual machines for programming languages.

**4. Ahead-of-time compilers (AOT):**

* It translates the source code into machine code before the program is executed.
* This is in contrast to JIT compilers, which generate machine code at runtime.
* They are commonly used for native applications, such as those developed for desktop and mobile platforms. (gaming or scientific computing).

**5. cross compilers:**

* generates machine code for a different platform than the one on which the compiler is running. For example, a cross-compiler running on a Windows PC can generate machine code for a Linux-based target platform.
* They are commonly used in embedded systems, where the target platform may have limited resources, or in software development for multiple platforms.

# 3rd: project analysis

 1**st phase Lexical Analysis**

Tokenizes the source code, removing whitespace and comments, use of regular expressions or token generators.

 2**nd phase Syntax Analysis**

It takes tokens as input and generates a parse tree as output, the parser checks that the expression made by the tokens is syntactically correct or not.

 3rd **phase** **Semantic Analysis**

It checks whether the parse tree follows the rules of language, keeps track of identifiers, their types and expressions.

 4th **phase** **Intermediate Code Generation**

It generates the source code into the intermediate code. It is between the high-level language and the machine language. should be generated in easily way to translate it into the target machine code.

 5th **phase** **Code Optimization**

improve the intermediate code so output could run faster and take less space. It removes the unnecessary lines and arranges the sequence of statements in order to speed up the program execution

6th **phase** **Code Generation**

 It takes the optimized intermediate code as input and maps it to the target machine language. Produces target machine code assembly-like output.

# 4th: CFG Design

**Non-Terminals**

* **E**: Expression
* **T**: Term
* **F**: Factor
* **Add\_op**:add operator,subtraction
* **Mulop**:multiply

**Production Rules**

1. Exp → E Add\_op term

2. Exp→ term

4. Term → Term Mulop Factor

5. Term → Factor

5. Factor → (Exp) | id | num

6. Mulop → \*,/

7.add\_op → +,-

# 5th: project implementation

1st: put imports for some libraries

* re: used for regular expressions
* binary tree: to make binary tree in syntax and in semantic phases
* ipython.display and widgets : to support Gui

import re

from binarytree import Node

from IPython.display import display

import ipywidgets as widgets

2nd: define tokens, special symbols, regular expression

* Define variables as reserved words and special symbols.
* Define regular expressions as: Matches valid identifiers: a sequence starting with a letter or underscore, followed by letters, digits, or underscores, matches floating-point numbers, matches integers, matches comments that start with %% and are followed by letters.
* Define some functions takes token as parameter to see which is reserved, symbolic,
* Use **re.fullmatch** to check if the token matches the identifier pattern.

reserved\_words = ['if', 'then', 'else', 'end', 'repeat', 'until', 'read', 'write']

special\_symbols = ['+', '-', '\*', '/', '=', '<', '(', ')', ';', ':=', '{', '}']

identifier\_pattern = r'[a-zA-Z\_][a-zA-Z0-9\_]\*'

float\_pattern = r'\b\d+\.\d+\b'

int\_pattern = r'\b\d+\b'

comment\_pattern = r'%%[a-zA-Z]\*'

def is\_reserved\_word(token):

    return token in reserved\_words

def is\_special\_symbol(token):

    return token in special\_symbols

def is\_identifier(token):

    return re.fullmatch(identifier\_pattern, token)

def is\_float(token):

    return re.fullmatch(float\_pattern, token)

def is\_int(token):

    return re.fullmatch(int\_pattern, token)

def is\_comment(token):

    return re.fullmatch(comment\_pattern, token)

3rd: define lexical phase: Preprocess equation:

* To define lexical phase
* Change pi to 3.14
* Put\* before ()
* Tokenization: Extracts tokens from the equation using regular expressions. Matches identifiers, numbers, symbols, and operators.
* Initializes a dictionary for mapping unique identifiers to id
* Iterates over tokens and processes them based on their type: Reserved words and special symbols are directly added to the new expression. Numbers (floats or integers) are also directly added.
* For each identifier, checks if it’s already mapped to an id. If not, assigns a new id placeholder (id1, id2, etc.) and updates the mapping.

def preprocess\_equation(equation):

    equation = equation.replace('pi', '3.14')

    equation = equation.replace('PI', '3.14')

    equation = re.sub(r'(\d)(\()', r'\1 \* \2', equation)

    tokens = re.findall(r'[\w.]+|:=|[+\-\*/=<>();{}]', equation)

    unique\_identifiers = {}

    new\_expression = []

    id\_counter = 1

    for token in tokens:

        if is\_reserved\_word(token):

            new\_expression.append(token)

        elif is\_special\_symbol(token):

            new\_expression.append(token)

        elif is\_float(token) or is\_int(token):

            new\_expression.append(token)

        elif is\_identifier(token):

            if token not in unique\_identifiers:

                id\_name = f'id{id\_counter}'

                unique\_identifiers[token] = id\_name

                id\_counter += 1

            new\_expression.append(unique\_identifiers[token])

        elif is\_comment(token):

            continue

        else:

            raise ValueError(f"Unknown token: {token}")

    return " ".join(new\_expression), unique\_identifiers

4th: define postfix to infix and syntax tree

* Perform postfix function by using chunk yard algorithm
* takes the postfix expression and constructs binary syntax tree. Each node in the tree represents an operator or operand.
* Each element in the postfix expression becomes a Node in the tree.
* Push a new Node containing the operand onto the stack.

 Pop two nodes from the stack (the right and left operands).

 Create a new Node for the operator, making the popped nodes its children.

 push the new Node back onto the stack.

 After processing all tokens, the last remaining node in the stack is the root of the syntax tree.

def infix\_to\_postfix(expression):

    precedence = {'+': 1, '-': 1, '\*': 2, '/': 2, '=': 0}

    stack = []

    postfix = []

    tokens = expression.split()

    for token in tokens:

        if is\_float(token) or is\_int(token) or is\_identifier(token):

            postfix.append(token)

        elif token in precedence:

            while stack and precedence.get(stack[-1], -1) >= precedence[token]:

                postfix.append(stack.pop())

            stack.append(token)

        elif token == '(':

            stack.append(token)

        elif token == ')':

            while stack and stack[-1] != '(':

                postfix.append(stack.pop())

            stack.pop()

    while stack:

        postfix.append(stack.pop())

    return postfix

def build\_syntax\_tree(postfix):

    stack = []

    for token in postfix:

        if is\_float(token) or is\_int(token) or is\_identifier(token):

            stack.append(Node(token))

        else:

            right = stack.pop()

            left = stack.pop()

            node = Node(token)

            node.left = left

            node.right = right

            stack.append(node)

    return stack.pop()

5th: define the semantic tree

* **has\_float**: Determines if the postfix expression contains any floating-point numbers

**Iterating Through Tokens**:

* For each token in the postfix expression:
  + **Operand (value or identifier)**:  
    If the token is a number (integer/float) or an identifier (e.g., variable name):
    - A new node is created for the token (Node(token)).
    - **Type Conversion**:  
      If the expression has floats and the token is an integer, an implicit **int-to-float conversion** node (inttofloat) is created. This ensures consistent types in operations. The conversion node becomes the parent of the integer node.

def build\_semantic\_tree(postfix):

    stack = []

    has\_float = any(is\_float(token) for token in postfix)

    for token in postfix:

        if is\_float(token) or is\_int(token) or is\_identifier(token):

            node = Node(token)

            if has\_float and is\_int(token):

                inttofloat\_node = Node("inttofloat")

                inttofloat\_node.left = node

                stack.append(inttofloat\_node)

            else:

                stack.append(node)

        else:

            right = stack.pop()

            left = stack.pop()

            operator\_node = Node(token)

            operator\_node.left = left

            operator\_node.right = right

            stack.append(operator\_node)

    return stack.pop()

6th: define intermediate code generator

 **temp\_counter**: A counter used to generate unique temporary variable names (Temp1, Temp2, etc.).

 **intermediate\_code**: A list to store the generated intermediate code instructions.

 **traverse(node)**: A recursive helper function that processes the semantic tree node by node.

 **nonlocal temp\_counter**: Ensures that temp\_counter is shared across all recursive calls.

def generate\_intermediate\_code(semantic\_tree):

    temp\_counter = 1

    intermediate\_code = []

    def traverse(node):

        nonlocal temp\_counter

        if node is None:

            return None

        if node.value == "inttofloat":

            operand = traverse(node.left)

            temp\_var = f"Temp{temp\_counter}"

            temp\_counter += 1

            intermediate\_code.append(f"{temp\_var} = inttofloat({operand})")

            return temp\_var

        if node.value == "=":

             right = traverse(node.right)

             intermediate\_code.append(f"id1 = {right}")

             return right

        if node.value in ['+', '-', '\*', '/', '<', '>']:

            left = traverse(node.left)

            right = traverse(node.right)

            temp\_var = f"Temp{temp\_counter}"

            temp\_counter += 1

            intermediate\_code.append(f"{temp\_var} = {left} {node.value} {right}")

            return temp\_var

        return node.value

    root\_result = traverse(semantic\_tree)

    return intermediate\_code

7th: define the code optimizer

 **optimized\_code**: An empty list to store the optimized instructions.

 **temp\_map**: A dictionary to track mappings of old temporary variables (TempX) to their optimized replacements.

 **new\_temp\_counter**: A counter for creating new temporary variable names (Temp1, Temp2, etc.) during optimization.

**Loop Over Intermediate Code**

python

Copy code

for i, line in enumerate(intermediate\_code):

* **Purpose**: Iterates through each instruction in intermediate\_code.
* **i**: The index of the current line in the list.
* **line**: The current intermediate code instruction being processed

def code\_optimizer(intermediate\_code):

    optimized\_code = []

    temp\_map = {}

    new\_temp\_counter = 1

    for i, line in enumerate(intermediate\_code):

        if "inttofloat" in line:

            temp\_var, expression = line.split(" = ")

            number = expression.replace("inttofloat(", "").replace(")", "").strip()

            temp\_map[temp\_var] = number

        else:

            for old\_temp, optimized\_value in temp\_map.items():

                line = line.replace(old\_temp, optimized\_value)

            if "Temp" in line:

                temp\_var, operation = line.split(" = ", 1)

                if i == len(intermediate\_code) - 1:

                    line = f"id1 = {operation.strip()}"

                else:

                    new\_temp\_var = f"Temp{new\_temp\_counter}"

                    temp\_map[temp\_var.strip()] = new\_temp\_var

                    line = f"{new\_temp\_var} = {operation.strip()}"

                    new\_temp\_counter += 1

            optimized\_code.append(line)

    return optimized\_code

8th: generate code generator

def generate\_code\_generator(optimized\_code):

    # To hold the assembly-like code

    assembly\_code = []

    reg1 = "r1"

    reg2 = "r2"

    for line in optimized\_code:

        line = line.strip()

        if not line:

            continue

        if "=" in line:

            lhs, rhs = line.split("=", 1)

            lhs = lhs.strip()

            rhs = rhs.strip()

            # Check for arithmetic operations

            if any(op in rhs for op in ["+", "-", "\*", "/"]):

                tokens = re.split(r'(\+|\-|\\*|/)', rhs)

                # Handle the first operand

                first\_operand = tokens[0].strip()

                if not first\_operand.isdigit():

                 assembly\_code.append(f"ldf {reg2}, {first\_operand}")

                # Process the remaining tokens

                current\_register = reg1

                for i in range(1, len(tokens), 2):

                    operator = tokens[i]

                    operand = tokens[i + 1].strip()

                    # Perform the operation

                    if operator == "+":

                        assembly\_code.append(f"addf {current\_register}, {current\_register}")

                    elif operator == "-":

                        assembly\_code.append(f"subf {current\_register}, {current\_register}")

                    elif operator == "\*":

                        assembly\_code.append(f"mulf {current\_register}, {current\_register}")

                    elif operator == "/":

                        assembly\_code.append(f"divf {current\_register}, {current\_register}")

                # Store the final result in the left-hand side variable

                if lhs.startswith("id"):

                    assembly\_code.append(f"stf {lhs}, {current\_register}")

            else:

                # Handle simple assignments

                assembly\_code.append(f"ldf {reg1}, {rhs}")

                assembly\_code.append(f"stf {lhs}, {reg1}")

    return assembly\_code

9th: create Gui and main function

* Widgets Setup
* A text input box for the user to enter their equation.
* description: A label for the box, displayed as “Input:”.
* placeholder: Placeholder text prompting the user to enter an equation.
* layout: Adjusts the width of the input box to 70% of the available space.
* Radio Buttons for Form Choice
* Output Sections
* Accordion to Group Output: Groups the output sections into an accordion widget, where each section can be expanded or collapsed.
* **Purpose**: Displays an error message in the lexical analysis section if no input is provided.

def main\_gui():

    input\_box = widgets.Text(

        description="Input:",

        placeholder="Enter your equation",

        layout=widgets.Layout(width="70%")

    )

    input\_choice = widgets.RadioButtons(

        options=["Math Form", "Source Form"],

        description="Form:",

        style={"button\_color": "lightgray"},

        layout=widgets.Layout(width="50%")

    )

    submit\_button = widgets.Button(description="Submit")

    lexical\_output = widgets.Output()

    syntax\_tree\_output = widgets.Output()

    semantic\_tree\_output = widgets.Output()

    intermediate\_code\_output = widgets.Output()

    optimized\_code\_output = widgets.Output()

    machine\_code\_output = widgets.Output()

    accordion = widgets.Accordion(children=[

        lexical\_output,

        syntax\_tree\_output,

        semantic\_tree\_output,

        intermediate\_code\_output,

        optimized\_code\_output,

        machine\_code\_output

    ])

    accordion.set\_title(0, "Lexical Analysis")

    accordion.set\_title(1, "Syntax Tree")

    accordion.set\_title(2, "Semantic Tree")

    accordion.set\_title(3, "Intermediate Code")

    accordion.set\_title(4, "Optimized Code")

    accordion.set\_title(5, "Machine Code")

    def on\_submit(\_):

        user\_input = input\_box.value

        form\_choice = input\_choice.value

        if not user\_input.strip():

            with lexical\_output:

                print("Error: Input cannot be empty.")

            return

        try:

            if form\_choice == "Math Form":

                modified\_expression, lexical\_result = preprocess\_equation(user\_input)

            elif form\_choice == "Source Form":

                if 'pi' in user\_input.lower():

                    with lexical\_output:

                        print("Lexical Error: 'pi' is not allowed in Source Form.")

                    return

                    modified\_expression, lexical\_result = preprocess\_equation(user\_input)

            lexical\_output.clear\_output()

            with lexical\_output:

                print("lexical analyzer:", modified\_expression)

                print("Identifiers Mapping:", lexical\_result)

            postfix\_expression = infix\_to\_postfix(modified\_expression)

            syntax\_tree = build\_syntax\_tree(postfix\_expression)

            syntax\_tree\_output.clear\_output()

            with syntax\_tree\_output:

                print("Postfix Expression:", ' '.join(postfix\_expression))

                print("\nSyntax Tree:")

                print(syntax\_tree)

            semantic\_tree = build\_semantic\_tree(postfix\_expression)

            semantic\_tree\_output.clear\_output()

            with semantic\_tree\_output:

                print("\nSemantic Tree")

                print(semantic\_tree)

       # Call the function to generate intermediate code

            intermediate\_code = generate\_intermediate\_code(semantic\_tree)

# Clear the output to avoid any cached results

            intermediate\_code\_output.clear\_output(wait=True)

# Print the intermediate code

            with intermediate\_code\_output:

             print("\nIntermediate Code:")

             for line in intermediate\_code:

              print(line)

            optimized\_code = code\_optimizer(intermediate\_code)

            optimized\_code\_output.clear\_output()

            with optimized\_code\_output:

                print("\nOptimized Code:")

                for line in optimized\_code:

                    print(line)

            machine\_code = generate\_code\_generator(optimized\_code)

            machine\_code\_output.clear\_output()

            with machine\_code\_output:

                print("\n Code generator")

                for line in machine\_code:

                    print(line)

        except Exception as e:

            lexical\_output.clear\_output()

            with lexical\_output:

                print(f"Error: {str(e)}")

    submit\_button.on\_click(on\_submit)

    display(widgets.VBox([

        widgets.HBox([input\_box, input\_choice]),

        submit\_button,

        accordion

    ]))

main\_gui()

    submit\_button.on\_click(on\_submit)

    display(widgets.VBox([

        widgets.HBox([input\_box, input\_choice]),

        submit\_button,

        accordion

    ]))

main\_gui()

# 6th: run of code



# 7th: conclusion

**At the end we know how compilers work by breaking down each phase,**

**analyzing code to generating machine instructions.**

**In the future, we could expand the compiler to support more**

**complex programming constructs and optimize performance further**

**“The most dangerous phrase in the language is, 'We've always done it this way.'" By learning compiler design, we challenge and innovate the way code is translated and executed. “**

**By: Grace Hopper**

# 8th: References

* <https://www.javatpoint.com/compiler-phases>
* <https://www.techtarget.com/whatis/definition/compiler#:~:text=A%20compiler%20is%20a%20special,bytecode%20or%20another%20programming%20language>
* <https://medium.com/@grp-21/types-of-compilers-ac2e9f5ab4cd>