University of Asia Pacific
Department of Computer Science & Engineering

Course Title: Compiler Design Lab

Course Code: CSE 430

Mini compiler Project Report

Submitted by:-

Junnatul Mawa

Section:-B1

Registration:- 20101070

Table of Contents:

Introduction..3

Compiler...3

Design..4

Diagram of the Phases:-...4

Lexical Analysis...5

Syntax Analysis..5

Semantic Analysis..5

Intermediate Code Generation...5

Code Optimization...5

Code Generation..6

Implementation...7

Result-...13

Reference-..14

Introduction

Compiler

A compiler is a program that can read a program in one language - the source language - and translate it to an equivalent program in another language - the target language. An important role of the compiler is to detect any errors in the source program during the translation process.

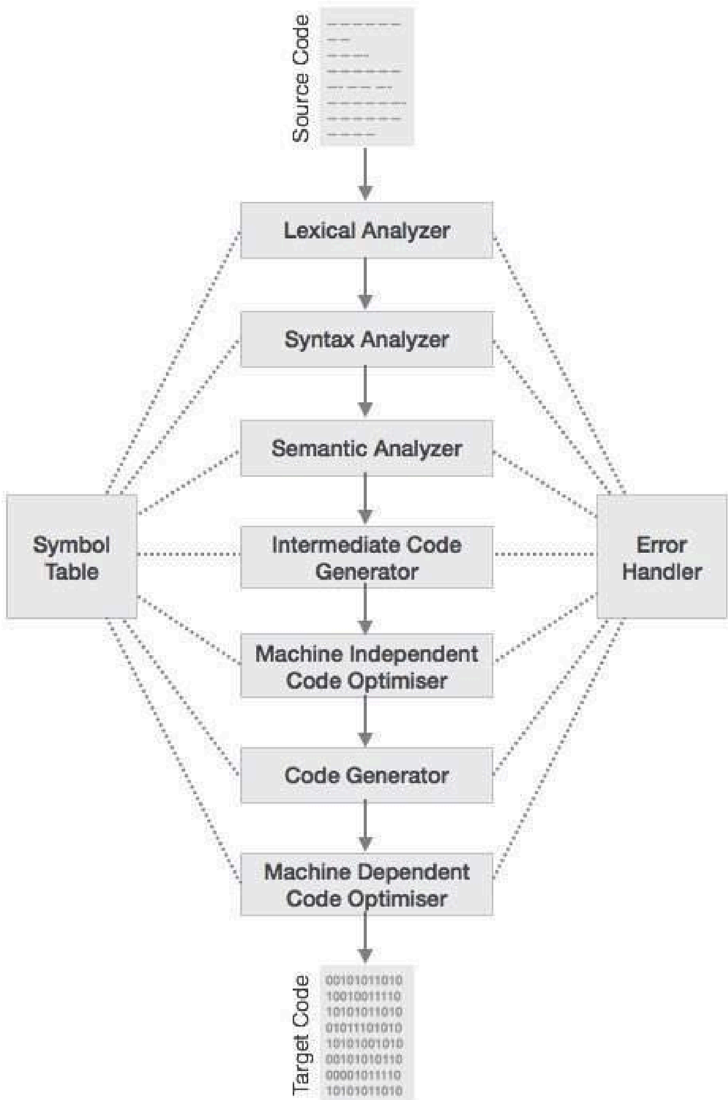
Structure of a compiler

There are two parts involved in the translation of a program in the source language into a semantically equivalent target program: analysis and synthesis. The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part. The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the front end of the compiler and the synthesis part is called as the back end

Design

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.

Diagram of the Phases:-



Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

```
<token-name, attribute-value>
```

Syntax Analysis

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

Implementation

```
import re

# Token types
TOKEN_TYPES = [
    ('NUMBER', r'\d+(\.\d*)?'),   # Integer or decimal number
    ('ADD', r'\+'),               # Addition
    ('SUB', r'-'),                # Subtraction
    ('MUL', r'\*'),               # Multiplication
    ('DIV', r'/'),                # Division
    ('LPAREN', r'\('),            # Left parenthesis
    ('RPAREN', r'\)'),            # Right parenthesis
    ('ID', r'[a-zA-Z_]\w*'),      # Identifiers
    ('ASSIGN', r'='),             # Assignment operator
    ('WHITESPACE', r'\s+'),       # Whitespace
]

# Tokenizer
def tokenize(code):
    tokens = []
    while code:
        match = None
        for token_type, pattern in TOKEN_TYPES:
            regex = re.compile(pattern)
            match = regex.match(code)
            if match:
                text = match.group(0)
                if token_type != 'WHITESPACE':  # Ignore whitespace
                    tokens.append((token_type, text))
                code = code[len(text):]
                break
```

```
        if not match:
            raise SyntaxError(f"Unexpected character: {code[0]}")
    return tokens

# Example usage
code = "a = 3 + 5 * (2 - 8) / 2"
tokens = tokenize(code)
print(tokens)

class ASTNode:
    def __init__(self, type, value=None, children=None):
        self.type = type
        self.value = value
        self.children = children if children else []

def parse(tokens):
    def parse_expression(index):
        node, index = parse_term(index)
        while index < len(tokens) and tokens[index][0] in ('ADD', 'SUB'):
            op = tokens[index]
            index += 1
            right_node, index = parse_term(index)
            node = ASTNode(op[0], op[1], [node, right_node])
        return node, index

    def parse_term(index):
        node, index = parse_factor(index)
        while index < len(tokens) and tokens[index][0] in ('MUL', 'DIV'):
            op = tokens[index]
            index += 1
            right_node, index = parse_factor(index)
            node = ASTNode(op[0], op[1], [node, right_node])
        return node, index

    def parse_factor(index):
```

```
        token = tokens[index]
        if token[0] == 'NUMBER':
            node = ASTNode('NUMBER', token[1])
            return node, index + 1
        elif token[0] == 'ID':
            node = ASTNode('ID', token[1])
            return node, index + 1
        elif token[0] == 'LPAREN':
            index += 1
            node, index = parse_expression(index)
            if tokens[index][0] != 'RPAREN':
                raise SyntaxError("Expected ')'")
            return node, index + 1
        else:
            raise SyntaxError(f"Unexpected token: {token}")

    def parse_assignment(index):
        if tokens[index][0] == 'ID' and tokens[index + 1][0] == 'ASSIGN':
            var_name = tokens[index][1]
            index += 2
            expr_node, index = parse_expression(index)
            return ASTNode('ASSIGN', var_name, [expr_node]), index
        return parse_expression(index)

    ast, index = parse_assignment(0)
    if index != len(tokens):
        raise SyntaxError("Unexpected tokens at the end")
    return ast

# Example usage
ast = parse(tokens)
print(ast)

def check_semantics(ast, symbol_table=None):
    if symbol_table is None:
```

```
        symbol_table = {}

    if ast.type == 'ASSIGN':
        var_name = ast.value
        expr_node = ast.children[0]
        value = evaluate_expression(expr_node, symbol_table)
        symbol_table[var_name] = value
        return symbol_table
    else:
        evaluate_expression(ast, symbol_table)
    return symbol_table

def evaluate_expression(node, symbol_table):
    if node.type == 'NUMBER':
        return float(node.value)
    elif node.type == 'ID':
        if node.value not in symbol_table:
            raise NameError(f"Undefined variable: {node.value}")
        return symbol_table[node.value]
    elif node.type in ('ADD', 'SUB', 'MUL', 'DIV'):
        left_val = evaluate_expression(node.children[0], symbol_table)
        right_val = evaluate_expression(node.children[1], symbol_table)
        if node.type == 'ADD':
            return left_val + right_val
        elif node.type == 'SUB':
            return left_val - right_val
        elif node.type == 'MUL':
            return left_val * right_val
        elif node.type == 'DIV':
            return left_val / right_val
    else:
        raise ValueError(f"Unexpected node type: {node.type}")

# Example usage
symbol_table = check_semantics(ast)
```

```
print(symbol_table)

def generate_ir(ast):
    ir_code = []

    def traverse(node):
        if (node.type == 'ASSIGN'):
            var_name = node.value
            expr_code = traverse(node.children[0])
            ir_code.append(f"{var_name} = {expr_code}")
            return var_name
        elif (node.type == 'NUMBER'):
            return node.value
        elif (node.type == 'ID'):
            return node.value
        elif (node.type in ('ADD', 'SUB', 'MUL', 'DIV')):
            left_code = traverse(node.children[0])
            right_code = traverse(node.children[1])
            temp_var = f"t{len(ir_code)}"
            ir_code.append(f"{temp_var} = {left_code} {node.value} {right_code}")
            return temp_var

    traverse(ast)
    return ir_code

# Example usage
ir_code = generate_ir(ast)
print(ir_code)

def optimize_ir(ir_code):
    # Simple optimization example: remove unnecessary assignments
    optimized_code = []
    for line in ir_code:
        if not line.startswith("t"):
            optimized_code.append(line)
```

```
    return optimized_code

# Example usage
optimized_ir = optimize_ir(ir_code)
print(optimized_ir)

def generate_machine_code(ir_code):
    machine_code = []
    for line in ir_code:
        machine_code.append(line)  # In a real compiler, this would be translated to actual machine code
    return machine_code

# Example usage
machine_code = generate_machine_code(optimized_ir)
print(machine_code)
```

Result-

Output from the terminal-

```
PS E:\Compiler Code> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe "e:/Compiler Code/Final
_compiler_mawa (1).py"
[('ID', 'a'), ('ASSIGN', '='), ('NUMBER', '3'), ('ADD', '+'), ('NUMBER', '5'), ('MUL', '*'), ('LPAREN', '('), (
'NUMBER', '2'), ('SUB', '-'), ('NUMBER', '8'), ('RPAREN', ')'), ('DIV', '/'), ('NUMBER', '2')]
<__main__.ASTNode object at 0x0000019CEAD13610>
{'a': -12.0}
['t0 = 2 - 8', 't1 = 5 * t0', 't2 = t1 / 2', 't3 = 3 + t2', 'a = t3']
['a = t3']
['a = t3']
Tokens: [('ID', 'h'), ('ASSIGN', '='), ('NUMBER', '12'), ('ADD', '+'), ('NUMBER', '5'), ('MUL', '*'), ('LPAREN'
, '('), ('NUMBER', '2'), ('ADD', '+'), ('NUMBER', '8'), ('RPAREN', ')'), ('DIV', '/'), ('NUMBER', '2')]

Abstract Syntax Tree: <__main__.ASTNode object at 0x0000019CEAD1C250>

Symbol Table: {'h': 37.0}

Intermediate Code: ['t0 = 2 + 8', 't1 = 5 * t0', 't2 = t1 / 2', 't3 = 12 + t2', 'h = t3']

Optimized Intermediate Code: ['h = t3']

Machine Code: ['h = t3']
PS E:\Compiler Code>
```

Reference-

1. https://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm