**Course - System Programming and Compiler Construction (SPCC)**

|  |  |
| --- | --- |
| **UID** | 2021300108 |
| **Name** | Hatim Sawai |
| **Class and Batch** | TE Computer Engineering - Batch C |
| **Date** | 26-04-2024 |
| **Lab #** | 8 |
| **Aim** | Write a program to create optimize code using different code optimization techniques |
| **Objective** | Implement and evaluate the functionality of a two-pass assembler. Construct a symbol table, resolving forward references, and generating machine code. |
| **Theory** | **Compiler Pass[1]**  A Compiler pass refers to the traversal of a compiler through the entire program. Compiler passes are of two types: Single Pass Compiler, and Two Pass Compiler or Multi-Pass Compiler. These are explained as follows.  **Single Pass Compiler**  If we combine or group all the phases of compiler design in a single module known as a single pass compiler.    In the above diagram, there are all 6 phases are grouped in a single module, some points of the single pass compiler are as:   * A one-pass/single-pass compiler is a type of compiler that passes through the part of each compilation unit exactly once. * Single pass compiler is faster and smaller than the multi-pass compiler. * A disadvantage of a single-pass compiler is that it is less efficient in comparison with the multipass compiler. * A single pass compiler is one that processes the input exactly once, so going directly from lexical analysis to code generator, and then going back for the next read.   **Problems with Single Pass Compiler**   * We can not optimize very well due to the limited context of expressions. * As we can’t back up and process it again so grammar should be limited or simplified. * Command interpreters such as bash/sh/tcsh can be considered Single pass compilers, but they also execute entries as soon as they are processed.   **Two Pass Compiler[2]**  A two-pass assembler is a type of assembly language translator that processes the source code in two consecutive passes or phases. In the first pass, the assembler scans the entire source code, building a symbol table that contains information about labels, variables, and their memory locations. It also detects syntax errors and performs macro expansion if applicable. During the second pass, the assembler generates the actual machine code, using the information gathered in the first pass.  **Pass 1: Symbol Table Construction**  During the first pass, the assembler reads the source code line by line, analyzing each instruction and directive. It identifies and records the symbols (labels, variable names) along with their associated memory locations or addresses. Additionally, it resolves any forward references encountered, ensuring that all symbols are defined before they are used. The symbol table generated during this pass serves as a reference for the subsequent code generation phase.  **Pass 2: Code Generation**  In the second pass, the assembler utilizes the symbol table constructed in the first pass to generate the machine code instructions. It revisits each line of the source code, translating assembly instructions into their binary equivalents or relocatable machine code. This phase may also involve resolving any remaining unresolved symbols and generating appropriate relocation information if the target machine supports relocatable code.  **Advantages of a Two-Pass Assembler**   1. **Efficient Symbol Resolution**: By dividing the assembly process into two passes, a two-pass assembler can efficiently handle forward references, ensuring that all symbols are properly resolved. 2. **Error Detection:** The two-pass approach allows the assembler to detect syntax errors, undefined symbols, or other issues early in the assembly process, improving overall code quality and debugging. 3. **Macro Expansion:** Two-pass assemblers often support macro expansion, enabling the use of reusable code fragments and simplifying program development.   **Difference between One Pass and Two Pass Compiler[1]**   |  |  | | --- | --- | | **One pass Compiler** | **Two Pass Compiler** | | It performs Translation in one pass | It performs Translation in two pass | | It scans the entire file only once. | It requires two passes to scan the source file. | | It generates [Intermediate code](https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/) | It does not generate Intermediate code | | It is faster than two pass [assemble](https://www.geeksforgeeks.org/introduction-of-assembler/)r | It is slower than two pass assembler | | A loader is not required | A [loader](https://www.geeksforgeeks.org/loader-in-compiler-design/) is required. | | No object program is written. | A loader is required as the object code is generated. | | Perform some professing of assembler directives. | Perform processing of assembler directives not done in pass-1 | | The [data structure](https://www.geeksforgeeks.org/data-structures/) used are:  The [symbol table](https://www.geeksforgeeks.org/symbol-table-compiler/), literal table, pool table, and table of incomplete. | The data structure used are:  The symbol table, literal table, and pool table. | | These assemblers perform the whole conversion of assembly code to machine code in one go. | These assemblers first process the assembly code and store values in the opcode table and symbol table and then in the second step they generate the [machine code](https://www.geeksforgeeks.org/machine-instructions/) using these tables. | | Example: [C](https://www.geeksforgeeks.org/c-programming-language/) and Pascal uses One Pass Compiler. | Example: Modula-2 uses Multi Pass Compiler. | |
| **Implementation / Code** | **Python Code:**  **import sys**  **def RemoveSpaces(x):**  **if (x != " ") or (x != ", "):**  **return x**  **def RemoveCommas(x):**  **if x[-1] == ",":**  **return x[ : len(x) - 1]**  **else:**  **return x**  **def CheckLiteral(element):**  **if element[ : 2] == "='":**  **return True**  **else:**  **return False**  **def CheckSymbol(Elements):**  **global SymbolTable, Opcodes**  **if (len(Elements) > 1) and ([Elements[-1], None, None, "Variable"] not in SymbolTable) and (Elements[-1] != "CLA") and (Elements[-2] not in ["BRP", "BRN", "BRZ"]) and (Elements[-1][ : 2] != "='") and (Elements[-1][ : 3] != "REG") and (not Elements[-1].isnumeric()):**  **return True**  **else:**  **return False**  **def CheckLabel(Elements):**  **global SymbolTable, Opcodes**  **if (len(Elements) >= 2) and (Elements[1] in Opcodes):**  **if Elements[0] not in SymbolTable:**  **return True**  **else:**  **return False**  **Opcodes = ["CLA", "LAC", "SAC", "ADD", "SUB", "BRZ", "BRN", "BRP", "INP", "DSP", "MUL", "DIV", "STP", "DATA", "START"]**  **AssemblyOpcodes = {"CLA" : "0000", "LAC" : "0001", "SAC" : "0010", "ADD" : "0011", "SUB" : "0100", "BRZ" : "0101","BRN" : "0110",**  **"BRP" : "0111", "INP" : "1000", "DSP" : "1001", "MUL" : "1010", "DIV" : "1011", "STP" : "1100"}**  **SymbolTable = []**  **LiteralTable = []**  **Variables = []**  **Declarations = []**  **AssemblyCode = []**  **location\_counter = 0**  **stop\_found = False**  **end\_found = False**  **file = open("Assembly Code Input.txt", "rt")**  ***# ERROR 1 : Checking for missing START statement***  **for line in file:**  ***# Checking for comments***  **if line[ : 2] != "//":**  **if line.strip() != "START":**  **print("STARTError : 'START' statement is missing. " + "( Line " + str(location\_counter) + " )")**  **sys.exit(0)**  **else:**  **file.seek(0, 0)**  **break**  ***# First Pass***  **for line in file:**  ***# Checking for comments***  **if line[ : 2] != "//":**  **Elements = line.strip().split(" ")**  **Elements = list(filter(RemoveSpaces, Elements))**  **Elements = list(map(RemoveCommas, Elements))**  ***# Removing comments***  **for i in range(len(Elements)):**  **if Elements[i][ : 2] == "//":**  **Elements = Elements[ : i]**  **break**  ***# ERROR 2 : Checking for too many operands***  ***# If the instruction doesn't contain a Label***  **if (len(Elements) >= 3) and (Elements[0] in Opcodes):**  **print("TooManyOperandsError : Too many operands used for the '" + Elements[0] + "' assembly opcode. " + "( Line " + str(location\_counter) + " )")**  **sys.exit(0)**  ***# If the instruction contains a Label***  **elif (len(Elements) >= 4) and (Elements[1] in Opcodes):**  **print("TooManyOperandsError : Too many operands used for the '" + Elements[1] + "' assembly opcode. " + "( Line " + str(location\_counter) + " )")**  **sys.exit(0)**  ***# ERROR 3 : Checking for less operands***  ***# If the instruction doesn't contain a Label***  **if (len(Elements) == 1) and (Elements[0] in ["LAC", "SAC", "ADD", "SUB", "BRZ", "BRN", "BRP", "INP", "DSP", "MUL", "DIV"]):**  **print("LessOperandsError : Less operands used for the '" + Elements[0] + "' assembly opcode. " + "( Line " + str(location\_counter) + " )")**  **sys.exit(0)**  ***# If the instruction contains a Label***  **elif (len(Elements) == 2) and (Elements[1] in ["LAC", "SAC", "ADD", "SUB", "BRZ", "BRN", "BRP", "INP", "DSP", "MUL", "DIV"]):**  **print("LessOperandsError : Less operands used for the '" + Elements[1] + "' assembly opcode. " + "( Line " + str(location\_counter) + " )")**  **sys.exit(0)**  ***# ERROR 4 : Checking for invalid opcodes***  **if stop\_found is False:**  **if len(Elements) == 3:**  ***# If the instruction contains a Label***  **if Elements[1] not in Opcodes:**  **print("InvalidOpcodeError : '" + Elements[1] + "' is an invalid opcode. " + "( Line " + str(location\_counter) + " )")**  **sys.exit(0)**  **if (len(Elements) == 2) and (Elements[1] == "CLA"):**  **pass**  **elif len(Elements) == 2:**  ***# If the instruction doesn't contain a Label***  **if Elements[0] not in Opcodes:**  **print("InvalidOpcodeError : '" + Elements[0] + "' is an invalid opcode. " + "( Line " + str(location\_counter) + " )")**  **sys.exit(0)**  ***# Check for STP***  **if (len(Elements) == 3) and (Elements[1] == "DATA"):**  **stop\_found = True**  ***# Check for END***  **if (len(Elements) == 1) and (Elements[0] == "END"):**  **end\_found = True**  **for i in range(len(LiteralTable)):**  **if LiteralTable[i][1] == -1:**  **LiteralTable[i][1] = location\_counter**  **location\_counter += 1**  **break**  **if not stop\_found:**  ***# Check for Literal***  **for x in Elements:**  **if CheckLiteral(x):**  **LiteralTable.append([x, -1])**  ***# Check for Labels***  **if CheckLabel(Elements):**  **SymbolTable.append([Elements[0], location\_counter, None, "Label"])**  ***# Check for Symbols***  **if CheckSymbol(Elements):**  **SymbolTable.append([Elements[-1], None, None, "Variable"])**  **elif stop\_found:**  **if (Elements[0] != "STP") and (Elements[0] != "END"):**  ***# ERROR 5 : Checking for multiple definations***  **if Elements[0] not in Variables:**  **Variables.append(Elements[0])**  **Declarations.append((Elements[0], Elements[2]))**  **else:**  **print("DefinationError : Variable '" + Elements[0] + "' defined multiple times. " + "( Line " + str(location\_counter) + " )")**  **sys.exit(0)**  ***# ERROR 6 : Checking for redundant declarations***  **if [Elements[0], None, None, "Variable"] not in SymbolTable:**  **print("RedundantDeclarationError : " + Elements[0] + " declared but not used.")**  **sys.exit(0)**  **location = SymbolTable.index([Elements[0], None, None, "Variable"])**  **SymbolTable[location][1] = location\_counter**  **SymbolTable[location][2] = Elements[2]**  **location\_counter += 1**  ***# ERROR 7 : Checking for missing END statement***  **if end\_found is False:**  **print("ENDError : 'END' statement is missing." + "( Line " + str(location\_counter) + " )")**  **sys.exit(0)**  ***# ERROR 8 : Checking for undefined variables***  **for x in SymbolTable:**  **if x[1] is None and x[3] == "Variable":**  **print("UndefinedVariableError : Variable '" + x[0] + "' not defined.")**  **sys.exit(0)**  ***# Printing Tables after First Pass***  **print(">>> Opcode Table <<<\n")**  **print("ASSEMBLY OPCODE OPCODE")**  **print("--------------------------")**  **for key in AssemblyOpcodes:**  **print(key.ljust(20) + AssemblyOpcodes[key].ljust(6))**  **print("--------------------------")**  **print("\n>>> Literal Table <<<\n")**  **print("LITERAL ADDRESS")**  **print("-------------------")**  **for i in LiteralTable:**  **print(i[0].ljust(12) + str(i[1]).ljust(7))**  **print("-------------------")**  **print("\n>>> Symbol Table <<<\n")**  **print("SYMBOL ADDRESS VALUE TYPE")**  **print("----------------------------------------------")**  **for i in SymbolTable:**  **print(i[0].ljust(16) + str(i[1]).ljust(12) + str(i[2]).ljust(10) + i[3].ljust(10))**  **print("----------------------------------------------")**  **print("\n>>> Data Table <<<\n")**  **print("VARIABLES VALUE")**  **print("-------------------")**  **for i in Declarations:**  **print(i[0].ljust(14) + str(i[1]).ljust(10))**  **print("-------------------\n")**  ***# Second Pass***  **file.seek(0, 0)**  **print(">>> MACHINE CODE <<<\n")**  **for line in file:**  ***# Checking for comments***  **if line[ : 2] != "//":**  **Elements = line.strip().split(" ")**  **Elements = list(filter(RemoveSpaces, Elements))**  **Elements = list(map(RemoveCommas, Elements))**  **s = ""**  ***# Removing comments***  **for i in range(len(Elements)):**  **if Elements[i][ : 2] == "//":**  **Elements = Elements[ : i]**  **break**  ***# To terminate machine code conversion***  **if (len(Elements) == 3) and (Elements[1] == "DATA"):**  **break**  **if Elements[0] == "STP":**  **AssemblyCode.append("00 "+ AssemblyOpcodes["STP"] + " 00 00 00")**  **print("00 " + AssemblyOpcodes["STP"] + " 00 00 00")**  ***# If the CLA opcode has a Label before it***  **elif (len(Elements) == 2) and (Elements[1] == "CLA"):**  **for i in range(len(SymbolTable)):**  **if Elements[0] == SymbolTable[i][0]:**  **AssemblyCode.append(str(SymbolTable[i][1]).rjust(2, "0") + " " + AssemblyOpcodes["CLA"] + " 00 00 00")**  **print(str(SymbolTable[i][1]).rjust(2, "0") + " "+ AssemblyOpcodes["CLA"] + " 00 00 00")**  **elif Elements[0] != "START":**  **if (len(Elements) == 1) and (Elements[0] == "CLA"):**  **AssemblyCode.append("00 " + AssemblyOpcodes["CLA"] + " 00 00 00")**  **print("00 " + AssemblyOpcodes["CLA"] + " 00 00 00")**  ***# If there is no Label***  **elif (len(Elements) == 2) and (Elements[1] != "CLA"):**  **print("00 " + AssemblyOpcodes[Elements[0]], end = " ")**  **s = "00 " + AssemblyOpcodes[Elements[0]] + " "**  ***# Dealing with Literals***  **if CheckLiteral(Elements[1]):**  **for i in range(len(LiteralTable)):**  **if LiteralTable[i][0] == Elements[1]:**  **AssemblyCode.append(s + "00 00 " + str(LiteralTable[i][1]).rjust(2, "0"))**  **print("00 00 " + str(LiteralTable[i][1]).rjust(2, "0"))**  ***# Dealing with Lables (BRP, BRZ, BRN)***  **elif Elements[0] in ["BRP", "BRN", "BRZ"]:**  **for i in range(len(SymbolTable)):**  **if SymbolTable[i][0] == Elements[1]:**  **AssemblyCode.append(s + str(SymbolTable[i][1]).rjust(2, "0") + " 00 00")**  **print(str(SymbolTable[i][1]).rjust(2, "0") + " 00 00")**  ***# Dealing with Registers***  **elif Elements[1][ : 3] == "REG":**  **AssemblyCode.append(s + "00 " + Elements[1][-1].rjust(2, "0") + " 00")**  **print("00 " + Elements[1][-1].rjust(2, "0") + " 00")**  ***# Dealing with Variables***  **else:**  **for i in range(len(SymbolTable)):**  **if SymbolTable[i][0] == Elements[1]:**  **AssemblyCode.append(s + "00 00 " + str(SymbolTable[i][1]).rjust(2, "0"))**  **print("00 00 " + str(SymbolTable[i][1]).rjust(2, "0"))**  ***# If the instruction conatins a Label***  **elif len(Elements) == 3:**  **for i in range(len(SymbolTable)):**  **if SymbolTable[i][0] == Elements[0]:**  **print(str(SymbolTable[i][1]).rjust(2, "0") + " " + AssemblyOpcodes[Elements[1]], end = " ")**  **s = str(SymbolTable[i][1]).rjust(2, "0") + " " + AssemblyOpcodes[Elements[1]] + " "**  ***# Dealing with Literals***  **if CheckLiteral(Elements[2]):**  **for i in range(len(LiteralTable)):**  **if LiteralTable[i][0] == Elements[2]:**  **AssemblyCode.append(s + "00 00 " + str(LiteralTable[i][1]).rjust(2, "0"))**  **print("00 00 " + str(LiteralTable[i][1]).rjust(2, "0"))**  ***# Dealing with Lables (BRP, BRZ, BRN)***  **elif Elements[1] in ["BRP", "BRN", "BRZ"]:**  **for i in range(len(SymbolTable)):**  **if SymbolTable[i][0] == Elements[2]:**  **AssemblyCode.append(s + str(SymbolTable[i][1]).rjust(2, "0") + " 00 00")**  **print(str(SymbolTable[i][1]).rjust(2, "0") + " 00 00")**  ***# Dealing with Registers***  **elif Elements[2][ : 3] == "REG":**  **AssemblyCode.append(s + "00 " + Elements[2][-1].rjust(2, "0") + " 00")**  **print("00 " + Elements[2][-1].rjust(2, "0") + " 00")**  ***# Dealing with Variables***  **else:**  **for i in range(len(SymbolTable)):**  **if SymbolTable[i][0] == Elements[2]:**  **AssemblyCode.append(s + "00 00 " + str(SymbolTable[i][1]).rjust(2, "0"))**  **print("00 00 " + str(SymbolTable[i][1]).rjust(2, "0"))**  **file.close()**  **file = open("MachineCode.txt", "x")**  **file.write("------------\nMACHINE CODE\n------------\n\n")**  **for x in AssemblyCode:**  **file.write(x + "\n")**  **file.close()**  **Assesmbly Code Input.txt:**  **START**  **// Comment Number One**  **// Comment Number Two**  **LoopOne CLA X**  **LAC A**  **ADD ='1'**  **SUB ='35'**  **Loop BRP Subtraction // Comment Number Three**  **Subtraction SUB ='5'**  **ADD B // Comment Number Four**  **MUL C**  **SUB D**  **MUL ='600'**  **BRZ Zero1 // Comment Number Five**  **Division DIV E**  **CLA**  **LAC REG1**  **BRP Positive**  **Zero SAC X**  **DSP X**  **STP**  **Positive CLA**  **DSP REG1**  **DSP REG2**  **A DATA 250**  **B DATA 125**  **C DATA 90**  **D DATA 88**  **E DATA 5**  **X DATA 0**  **END** |
| **Output** |  |
| **Conclusion** | In conclusion, the experiment demonstrated the effectiveness of a two-pass assembler in efficiently constructing symbol tables, resolving forward references, and generating machine code. The approach facilitated early error detection, contributing to improved code reliability and debugging. While supporting macro expansion and enabling reusable code fragments, the two-pass method may have limitations in terms of increased memory usage and processing time, particularly for complex programs. Nonetheless, its benefits in symbol resolution and error detection make it a valuable tool for assembly language programming tasks, offering insights into optimizing software development processes. |
| **References** | [1] GeeksforGeeks: Single Pass and Two Pass Compilers  <https://www.geeksforgeeks.org/single-pass-two-pass-and-multi-pass-compilers/>  [2] ChatGPT (April 22, 2024) Code Generation <https://chat.openai.com/c/cc26cb94-bc49-4b94-9324-afcfb22f1742> |