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

|  |  |
| --- | --- |
| **UID** | 2021300108 |
| **Name** | Hatim Sawai |
| **Class and Batch** | TE Computer Engineering - Batch C |
| **Date** | 1-04-2024 |
| **Lab #** | 9 |
| **Aim** | Write a program to Implement a 2 pass Macro Processor |
| **Objective** | Implement Macros in Assembly language, and a 2 pass macro processor to pass the program to improve code efficiency and readability. |
| **Theory** | **Macros:**  Macros are a fundamental concept in programming that allows code to be written in a more abstract and reusable manner. They provide a way to define and use reusable code segments within a program. Macros are defined using a macro definition, and they are invoked or called using a macro invocation.  **Macro Definitions:**  A macro definition specifies the name of the macro and the sequence of instructions or expressions it represents. Macro definitions are typically defined using a special syntax or keyword. Here's an example of a macro definition in assembly language:  ADD MACRO a, b  MOV a, R0  ADD b, R0  MEND  In this example, ADD is the name of the macro, and a and b are the parameters.  **Macro Invocations:**  Macro invocations are instances where the macro is used or called within the code. When a macro is invoked, the macro processor replaces the macro invocation with the corresponding sequence of instructions or expressions defined in the macro definition. Here's an example of a macro invocation:  ADD A, B  **Macro Expansions:**  Macro expansion is the process of replacing macro invocations with their corresponding sequence of instructions or expressions. This process is performed by a macro processor before the code is passed to the actual compiler or interpreter for execution.  **Macro Processor:**  A macro processor is a program or part of a compiler that performs macro expansion. It takes the input code containing macro invocations, expands these macros, and produces the output code with the expanded macros.  **2 Pass Macro Processor:**  A 2-pass macro processor is a type of macro processor that performs macro expansion in two passes or phases.  **First Pass:**  In the first pass, the macro definitions are expanded.  The macro processor scans the entire source code and expands all macro invocations, replacing them with their corresponding sequence of instructions or expressions defined in the macro definition.  **Second Pass:**  In the second pass, the actual code is processed.  This pass is performed by the actual compiler or interpreter.  The code generated in the first pass, which contains expanded macros, is passed to the compiler or interpreter for further processing and execution.  **Advantages of 2 Pass Macro Processor:**  1. Provides efficient and flexible macro expansion.  2. Reduces the complexity of code by allowing the use of macros.  3. Improves code readability and maintainability by reducing code duplication.  4. Allows for modular programming by enabling the reuse of code fragments.  **Comparison:**  Below is a comparison between single pass and 2 pass macro processor: |
| **Implementation / Code** | **from prettytable import PrettyTable**  **class DefinitionTable:**  **def \_\_init\_\_(self):**  **self.index = None**  **self.definition = None**  **self.arg = [None, None]**  **self.next = None**  **class ArgumentListArray:**  **def \_\_init\_\_(self):**  **self.index = None**  **self.arg = None**  **self.next = None**  **class NameTable:**  **def \_\_init\_\_(self):**  **self.index = None**  **self.name = None**  **self.dt\_index = None**  **self.next = None**  **def find\_arg\_index(arg, al\_head):**  **temp = al\_head**  **while temp is not None:**  **if temp.arg == arg:**  **return temp**  **temp = temp.next**  **return None**  **def find\_name(name, nt\_head):**  **temp = nt\_head**  **while temp is not None:**  **if temp.name == name:**  **return temp.dt\_index**  **temp = temp.next**  **return None**  **def pass1(fp):**  **global MDTC, MNTC**  **MDTC = MNTC = 1**  **dt\_head = None**  **nt\_head = None**  **al\_head = None**  **al\_index = 1**  **while True:**  **line = fp.readline()**  **if not line:**  **break**  **if "MACRO" in line:**  **tokens = line.split()**  **print(f"\nMACRO {tokens[0]} Detected...\n")**  **if nt\_head is None:**  **nt\_head = NameTable()**  **nt\_temp = nt\_head**  **else:**  **nt\_temp.next = NameTable()**  **nt\_temp = nt\_temp.next**  **nt\_temp.index = MNTC**  **MNTC += 1**  **nt\_temp.name = tokens[0]**  **print(f"\n{tokens[0]} added into Name Table")**  **for token in tokens[1:]:**  **if token != "MACRO" and token != "\n":**  **if al\_head is None:**  **al\_head = ArgumentListArray()**  **al\_temp = al\_head**  **else:**  **al\_temp.next = ArgumentListArray()**  **al\_temp = al\_temp.next**  **al\_temp.index = al\_index**  **al\_index += 1**  **al\_temp.arg = token**  **print(f"\nArgument {al\_temp.arg} added into argument list array")**  **if dt\_head is None:**  **dt\_head = DefinitionTable()**  **dt\_temp = dt\_head**  **else:**  **dt\_temp.next = DefinitionTable()**  **dt\_temp = dt\_temp.next**  **dt\_temp.definition = nt\_temp.name**  **print(f"\nDefinition table entry created for {nt\_temp.name}")**  **nt\_temp.dt\_index = dt\_temp**  **while True:**  **line = fp.readline()**  **if line.strip() == "MEND":**  **break**  **tokens = line.split()**  **is\_arg = 0**  **index = 0**  **for token in tokens:**  **if is\_arg == 0:**  **if dt\_head is None:**  **dt\_head = DefinitionTable()**  **dt\_temp = dt\_head**  **else:**  **dt\_temp.next = DefinitionTable()**  **dt\_temp = dt\_temp.next**  **dt\_temp.index = MDTC**  **MDTC += 1**  **dt\_temp.definition = token**  **print(f"\nEntry appended for {dt\_temp.definition} at index {dt\_temp.index}")**  **is\_arg = 1**  **else:**  **if find\_arg\_index(token, al\_head) is None:**  **if al\_head is None:**  **al\_head = ArgumentListArray()**  **al\_temp = al\_head**  **else:**  **al\_temp.next = ArgumentListArray()**  **al\_temp = al\_temp.next**  **al\_temp.index = al\_index**  **al\_index += 1**  **al\_temp.arg = token**  **dt\_temp.arg[index] = al\_temp**  **else:**  **dt\_temp.arg[index] = find\_arg\_index(token, al\_head)**  **index += 1**  **# print("\nAll three tables are updated. Pass 1 Complete!\n")**  **# Assuming nt\_head, dt\_head, and al\_head are initialized in the main function**  **print\_name\_table(nt\_head)**  **print\_definition\_table(dt\_head)**  **print\_argument\_list\_array(al\_head)**  **def pass2(fp):**  **line = fp.readline()**  **while line:**  **print(line)**  **temp = find\_name(line, nt\_head)**  **if temp is not None:**  **while temp.definition != "MEND":**  **print("-", temp.definition, temp.arg[0], temp.arg[1])**  **temp = temp.next**  **line = fp.readline()**  **print("\nOutput file updated with expanded code. Pass 2 Complete!\n")**  **def print\_name\_table(nt\_head):**  **table = PrettyTable(["Index", "Name", "Definition Table Index"])**  **temp = nt\_head**  **while temp:**  **table.add\_row([temp.index, temp.name, temp.dt\_index.index])**  **temp = temp.next**  **print("Name Table:")**  **print(table)**  **def print\_definition\_table(dt\_head):**  **table = PrettyTable(["Index", "Definition", "Arguments", "Next"])**  **temp = dt\_head**  **while temp:**  **arg\_list = [arg.arg for arg in temp.arg if arg]**  **table.add\_row([temp.index, temp.definition, arg\_list, temp.next])**  **temp = temp.next**  **print("\nDefinition Table:")**  **print(table)**  **def print\_argument\_list\_array(al\_head):**  **table = PrettyTable(["Index", "Argument", "Next"])**  **temp = al\_head**  **while temp:**  **table.add\_row([temp.index, temp.arg, temp.next])**  **temp = temp.next**  **print("\nArgument List Array:")**  **print(table)**  **def main():**  **global nt\_head, al\_head**  **nt\_head = None**  **al\_head = None**  **try:**  **with open("input.asm", "r") as fp:**  **print("\nPass 1 in progress\n")**  **pass1(fp)**  **with open("input.asm", "r") as fp:**  **print("\nPass 2 in progress\n")**  **pass2(fp)**    **except IOError:**  **print("\nFailed to open the assembly file!")**  **if \_\_name\_\_ == "\_\_main\_\_":**  **main()** |
| **Output** |  |
| **Conclusion** | A 2 pass macro processor offers several advantages over a single pass macro processor, including improved efficiency, code optimization, and ease of maintenance. By performing macro expansion in two passes, it enables more efficient and flexible code processing, leading to faster processing times and improved code quality. |
| **References** | [1] Chatgpt, <https://chat.openai.com/share/2f45bddf-4a87-49ef-b4c0-c0d2fa2e4fb7> |