CS 202 A1

Dipeam Dasgupta (e0215188) Assignment 1.

Q1/ How will you classify a software as either an application software or a system software?

Based on purpose and functionality, softwares can be divided into 2 types basically.

Application software: These softwares are designed to perform specific tasks or provide solution to end users. It is created with intention to serve needs of individuals, busines and organizations. Some types include:

Productivity type : Word processors, spreadsheets, etc.
Multimedia type : photo editors, video editors, media player
communication type : Email clients, messaging applications.

System software : This software provides platform and set of essential services which are required to run a computer system properly. It is generally written by manufacturer or system administrator for easy use of system. Some examples include:

(I) The OS : primary system software.
(II) Compiler (IV) Linker (VI) Text editor.
(III) Assembler (V) Loader (VII) Debugger

OS - operating system.

Q2/ Why is "Assembler" considered as a system program?

Assembler is a tool to convert assembly language program into machine language one understandable by processor executing it. It plays crucial role in translation and execution of low-level machine code instructions.

Reasons for assembler being considered a system program:

(I) Low level programming language closely corresponding to the machine code instructions executed by CPU.

(II) Translates assembly code, written by using mnemonic instructions and other symbolic representations.

(III) It provides instructions that directly manipulate the hardware components of computer system, for ex: registers, memory addresses and i/o devices.

(IV) In some cases, assembler include additional functionalities like linking and loading.

(V) It closely integrated with OS and is included as part of development chain by OS. It works in combination of other system software components such as compilers, linker, loader to ensure efficient and proper execution of system programs & resources.

So, for the crucial role played in system maintenance and operation assemblers are considered as a system program

Q3/ Why it is useful to perform assembly process in multiple passes) Give an outline of division of activities among passes of multipass assembler.

It is useful and important for assembly process to perform in multiple passes. In multiple passes, symbol resolution, forward reference, operand calculation, code optimization, error detection are done.

Multiple pass assembly process solves the forward reference problem by using the first pass to generate a symbol table for all variables and references. In this method all errors are detected. Simple errors are detected in first pass and complex errors in second pass. Multiple pass also helps the assembler to optimize generated machine code. Complex operand calculations or resolving addressing modes are done more accurately in multiple passes. Finally, it leads to more efficient machine code generation.

In multiple passes, an assembler completes assembling generally in 2 passes. outline of the activities in 2 passes are given below:

Pass 1:

(i) Define symbols and literals and remember them in a symbol table and literal table respectively.

(ii) keeping track of location counter.

(iii) processing pseudo-operations.

(iv) Handling forward references where symbols used before being defined.

Pass 2:

(i) Generating object code by converting symbolic opcode into respective numeric opcode.

(ii) Generating data for literals and look for values of symbols.

(iii) Executing resolution of symbols.

(iv) Do error detecting and reporting.

Q4/ what are the activities performed in analysis and synthesis phase of assembler?

Assembler operates in two main phases which are analysis and synthesis respectively. Analysis phases does the work of validation, error check and symbol table creation. Synthesis phase does work of conversion into machine languge.

Analysis Phase

(I) Primary function includes building of the symbol table. It determines memory address with which each symbolic name used in a program is associated in assembly program.

(II) Address of N would be known only after fixing the addresses of all program elements. This is known as memory allocation. This is performed using location counter (LC)

(III) It ensures that the location counter always contains the address that next memory word in the target program should have.

(IV) At the start of its processing, it initializes the location counter to the constant specified in START statement

(V) while processing, it checks statement for label.

(vi) After label entered, it finds how many memory words are needed for instruction and updates address of LC

(vii) Amount of memory required is obtained for length field in mnemonics table

(viii) Notation <LC> is used for address contained in location counter.

(ix) Symbol table is constructed during analysis & used in synthesis.

Synthesis Phase

(i) Assembler translates each to assembly language instruction into its corresspondig machine code representation.

(ii) Calculation of values of operands used in assembly instructions

(iii) Handles in relocation, adjusts memory addresses based on final location of code in memory.

(iv) Resolves symbols, labels or variable names.

(v) Optionally perform optimization of codes.

(vi) performs error detection and handling. Checks for syntax error, invalid instructions during code generation process and reports them using error messages.

(vii) Produces the object file or output file containing compiled machine code.

So, these are activities in both phases makes assembler an essential tool for transforming assembly language to machine code.

Q5/ Compare the two features - macro and subroutine in a programming language.

Macros and subroutines are both programming constructs used for modular programming.

similarities

(I) Both allow for the reuse of code segments in a program; reduces duplication.

(II) Both encapsulate a sequence of instructions into single entity.

(III) Both can accept parameters or arguments that provide input to code segment being executed.

Differences:

| Macro | Subroutine |
|---|---|
| (I) Macro name in mnemonic field leads to expansion only | (I) Subroutine name in call statement leads to execution. |
| (II) statements of the macro body are expanded each time macro is invoked. | (II) statements of subroutine appear only once, regardless how many times it is called. |
| (III) Macros are completely handled by by assembler during assembly time | (III) subroutines are completely handled by hardware at runtime. |
| (IV) Macro defn and expansion are executed by assembler | (IV) Hardware executes subroutine call. |
| (V) Hardware knows nothing about macros | (V) Assembler knows nothing about subroutines. |

Q6/ What data structures may be used for macro expansion?

There are 3 main data structures which are involved in macro expansion process:

(i) DEFTAB (ii) NAMTAB (iii) ARGTAB.

All macro definitions in program are stored in DEFTAB. (include macro prototype and macro body statements). In NAMTAB macro names are entered. ARGTAB is used mainly in expansion of macro invocations. After recognition, arguments are stored in ARGTAB according to their position in argument list.

The macro names enters into NAMTAB which serves as an index to DEFTAB. For each macro instruction defined; NAMTAB contains pointers to the beginning and end of the definition DEFTAB. ARGTAB is the used for expansion of macro invocations. As the macro is expanded, arguments from ARGTAB are substituted for corresponding parameters in the macro body.

2021511188

Q7/ Why is linking required after a program is translated

Linking is required after a program is translated (compiled or assembled) for several reasons:

(i) During translation process, assigned temporary addresses of symbols. (function name, variables etc) are resolved by linking as it matches them with their actual memory addresses in other object file or libraries.

(ii) Linking combines individual and multiple source and object files, resolves external refrences and integrates the code and data from different modules to create single executable file.

(iii) Linking adjusts the relative addresses of object files to reflect final memory layout of the program. It also allocate memory segments for code, data, stack and other program sections.

(iv) Linking ensures the correct linking mechanism is applied based on program's requirements.

So, based on activities of linking, it can be said that it is crucial and important after a program is translated to create a complete and executable program.

Q8/ what does the term object file mean in context of program linking?. What are shared objects?

In context of program linking, an object file refers to a file that contains compiled or assembled code and data generated from source code or intermediate representations. An object file mainly contains machine code instructions, data definitions, symbol tables, relocation information and other metadata necessary for linking and loading. It is an intermediate representation of program that has been translated but has not been fully linked to create an executable file yet

Shared objects, also known as shared libraries or DLL, are a specific type of object file that can be dynamically linked and loaded by multiple programs or processes at runtime. Shared objects contain reusable code and data and thus reduces memory consumption. It provides a way to modularize code which result in efficient memory use and dynamic linking facility provides flexibility and allows program to dynamically load and use shared libraries based on specific runtime conditions.

Q9/ What are the stages ~~of~~ in the process of compilation?

The stages of process of compilation includes:
(i) lexical analysis (ii) syntax analysis (iii) semantic analysis
(iv) Intermediate code generation (v) Target code generation
(vi) symbols Table Management (vii) Error handling & recovery.

## Lexical Analysis

The source code divides into tokens such as keywords, identifiers, literals and operators. whitespace & comments are typically ignored.

## Syntax Analysis

The tokens from the lexical analysis stage are analyzed to determine syntactic structure of code. A parse tree or and abstract syntax tree is generated to represent structure of code

## Semantic Analysis

This phase is generally optional. Here the semantics and meaning of code are analyzed. Syntax tree of token with its attributes are made.

## Intermediate Code Generation:

Intermediate code such as three addren code /byte code is generated from AST. It represents simplified version of original code and is easier to optimize. It is different from machine code and doesnot require registers

## Code Optimization

It is also an optional phase, which executed if and only if intermediate code needs to be optimized. Optimization techniques eliminate redundent code, improve memory accen patterns, and reduce execution time

## Code Generation

Optimized intermediate code is translated to be machine code specific to the target hardware or virtual machine. Memory locations are selected for each of the variables.

## Symbol Table Manager

Symbol references are resolved as per symbol table and memory addresses are assigned to symbols. Relocation info is generated to allow executable code to be loaded at different memory locations.

## Error Detection

Each ~~phases~~ stages can encounter error, stages somehow deal with errors. syntax & semantic handle large amount of errors.

Q10/ How are 'expansion time variables' in macros different from normal program variables?

Expansion time variables, also known as macro local variables. are variables defined within a macro that are used during expansion of the macro. These are different from normal program variables by the ways:

(i) ETV ~~case~~ scopes are limited to macro definition itself. But ~~the~~ normal variables defined and accessed by whole block of function.

(ii) ETV exist during the expansion of macro only. Normal variables have defined lifetime based on scope and can persist throughout program execution.

(iii) ETV doesnot cause name conflicts as they are not visible outside macro expansion. But Normal variables can cause conflict if declared in multiple scopes.

(iv) ETV donot retain their values. Each time macro expanded, variables are reinitialized. Normal variables retain their values across different function calls.

So, in the following ways expansion time variables are different from normal program variables

202151183

Q11/ How are local labels handled in macro processing?

Local labels in macro processing are handled in a way that ensures their uniqueness and prevents conflicts with labels used outside of the macro.

Local labels are scoped to macro to which they are defined. They are not visible or accessible outside macro definition, ensuring no conflict with labels used in calling code.

Local labels often follow naming convention which distinguishes them from global variables. Also to ensure uniqueness, macro processors often maintain a counter and numbering scheme for local labels. Each time a macro is expanded, a new set of local labels is generated. Counter ensures each occurrence of a local label in the macro expansion is distinct.

During macro expansion, macro processor keeps track of context in which labels are encountered. When expanding macro, labels used within macro are transformed to ensure uniqueness within expanded code.

202151103

Q12/ what is position independent code? How does it help in solving the relocation problem?

Position independent code(PIC) is a type of machine code or executable code that can execute properly regardless of its memory location or address. It is designed to be relocatable meaning it can be loaded and executed at any memory location without requiring modification or adjusting data.

PIC is designed to be independent of absolute memory addresses as used by instructions and data. PIC code and data reference to offsets or relative addresses which allows them to relocate easily without modifications.

PIC code is also written in a way that instructions are self contained and don't rely on any absolute address. They use indirect or relative addressing to access data or jump to other parts of code which helps code to execute correctly.

Also, PIC is commonly used in dynamically linked libraries (shared objects) where multiple programs can dynamically load & link with same library code. PIC enables library code to be loaded at different memory locations, and thus resolving the relocation problem and allowing sharing of same code without conflicts.

Q13/ what are positional parameters, keyword parameters and expansion time variables in macros! Give an example

Positional parameters, keyword parameters, and expansion time variables are elements used in macros to customize and parameterize behaviour of macro expansions.

## Positional Parameters

These parameters in macro are specified by their position in order or order when macro is invoked. They allow arguments to be passed to the macro and used within its expansion. These values passed as arguments are substituted into macro expansion at corresponding positions of parameters.

Example:

```
#define ADD (a,b) ((a)+(b))
int sum = ADD (8,3); // Macro invocation with pos. param.
                     // Macro expands to ((8)+(3))
                     // value of sum becom 8. //
```

## Keyword Parameters.

These parameters in a macro are specified by their names when the macro is invoked. They allow arguments to be passed to the macro in any order using names of parameters to associate values with specific parameters. It provides flexibility and clarity when invoking macros with multiple arguments.

Example:

```
#define CIRCLE (rad, color)
  printf ("Circle with radius: %f ; color: %s \n", rad, color)

CIRCLE (color "red" ; radius: 25);
```

## Expansion Time Variables.

These are local variables defined within a macro that exist only during expansion of macro. Allows temporary storage and manipulation of values within macro expansion. These are scoped to macro and are not visible or accessible outside

Example

```
#define Print_Int (x) \
    do { \
        int temp = (x); \
        printf ("value: %d\n", temp); \
    } while (0)

int num = 10;
Print_Int (num);
```

output

```
value: 10
```

Q.14/ How can you load and call subroutines using dynamic linking?

Dynamic linking allows loading and calling of subroutines at runtime. It is often used to allow several executing program to share one copy of subroutine. It can avoid necenity of loading entire library for each execution.

Whenever user program requires subroutine for its execution, the program makes a load and call service request to OS. The parameter of that request is the symbolic name (ERRHANDL) of the routine to be called.

OS examines its internal tables to determine whether or not the routine is already loaded. If necessary, routine is loaded from specified user/system libraries. Control is then passed from OS to routine being called.

When subroutine (called) completes its processing it returns to its caller. OS then returns control to the program that issued the request. If a subroutine is still in memory, a second call to it may not require another load operation. Control may simply be passed from the dynamic loader to called routine.

So, in this way load and call can be done using dynamic linking.

Q15/ Explain Data structures used for macro processing. Write an algorithm for expansion (macro).

Data structures involved in macro processing are:

(i) DEFTAB (ii) NAMTAB (iii) ARGTAB.

DEFTAB

All macro definitions in program are stored in DEFTAB. including macro prototype and body statements. comment lines are not entered as they are not part of macro expansion. References to macro instruction parameters are converted to positional notation for efficiency in substituting arguments.

NAMTAB

It is also known as name table. The macro names are entered into NAMTAB. It contains pointers to beginning and end of definition in DEFTAB.

ARGTAB

It is also known as Argument Table. and it is used during expansion of macro invocations. when macro invocation statements are recognised, arguments are stored in ARGTAB according to their position in argument list. As the macro expanded, arguments from ARGTAB are substituted for corresponding parameters in macro body.

202151188.

A descriptive algorithm for macro expansion is as follows:

S(1) Initialize Macro Definition Table (MDT) with predefined macros.

S(2) Scanning source code, identifying macro invocations. If encountered,
create entry on Macro Invocation stack, sb
Store macro name and arguments.
If not to S5.

S(3) Look up macro definition in MDT. (macro name from invocation stack.

S(4) If macro definition found,
perform parameter substitution by replacing formal parameter with actual arguments provided in Macro invocation stack

S(4) Expand macro code by substituting tokens and token sequence as per macro definition. If nested macro invocations encounter, repeat step 3 to 4.

S(5) Complete macro expansion and replace original macro invocation in source code with expanded code from token buffer.

S(6) Continue scanning of source code for additional macro invocations and repeat until all are processed.

S(7) output the fully preprocessed source code where all macro invocations have been expanded.