UNIVERSITY INSTITUTE of COMPUTING

Asia's Fastest Growing University

System Software
CAT-204

Design By:

Prof. Pawandeep Sharma

A.P

Chandigarh University-Gharuan

UNIVERSITY INSTITUTE of COMPUTING
Asia's Fastest Growing University

Syllabus

UNIT-I

Introduction to System Software: Machine Structure, evolution of operating system, machine language.

Assembler: Elements of Assembly Language Programming, General design procedure, design of a Two Pass Assemblers, A Single Pass Assemblers Design.

Table Processing: Searching & Sorting.

UNIVERSITY INSTITUTE of COMPUTING
Asia's Fastest Growing University

Syllabus

UNIT-II

Macro and Macro Processors: Macro instructions, Features of a macro Facility: macro Instruction arguments, Conditional macro expansion, Macro calls within macros, Macro instruction defining macros, Advanced Macro Facilities, Implementation of simple macro processor, Two-pass algorithm, Implementation of macro calls within macros, Implementation within an assembler.

Linkers – Translated linked and load time addresses, relocation and linking concepts, Design of a linker, self relocating programs.

Syllabus

UNIT-III

Loaders: Loader scheme, absolute loaders, Subroutine linkages, Relocating loaders, Direct linking loaders, binders, linking loaders, overlays, Dynamic Binders, Design of an Absolute Loader, Design of a Direct-Linking Loader. Compilers: Phases of Compiler Construction, Symbol Table, Top-down and bottom-up Parsing, Operator-Precedence Parsing, LR Parsers, Code Generation and Code Optimization, Memory management, Design & other issues.

UNIVERSITY INSTITUTE *of*
COMPUTING
Asia's Fastest Growing University

Linkers

Definition of linker

- Linker is a system program that combines the code of target program with code of other programs and library routines.
- To facilitate linking the language translator build an object module for a program which contains both target code of the program and information about other programs and library routines that it need to invoke during execution.
- The linker extracts this information from the object module , locates the needed programs /routines and combines them with the target code of the program to produce a program in the machine language that can execute without requiring the assistance of any other program. such a program is called a binary program.

4 steps of program Execution

1. Translation of the program, which is performed by a processor called translator.

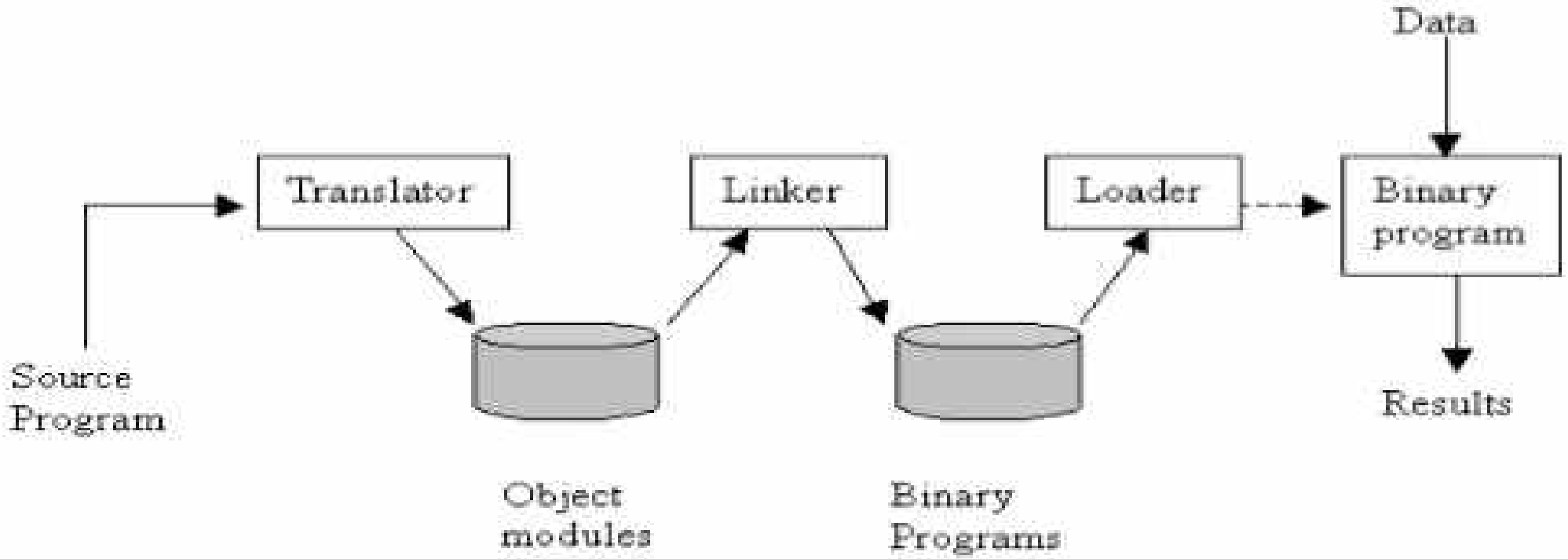
2. Linking of the program with other programs for execution, which is performed by a separate processor known as linker.

3. Relocation of the program to execute from the memory location allocated to it, which is performed by a processor called loader.

4. Loading of the program in the memory for its execution, which is performed by a loader.

4 steps of program Execution

Figure below shows a schematic flow of program execution, in which a translator, linker and loader performs functions such as translating and linking a program for the final result of the program.



Object Module

The object module of a program contains information, which is necessary to relocate and link the program with other programs.

The object module of a program, P, consists of various components:

1. **Header** contains translated origin, size and execution start address of the program P. Translated origin is the address of the origin assumed by the translator.
2. **Program** contains the machine language program corresponding to P.
3. **Relocation table(RELOCTAB)** maintains a list of entries that contains the translated address for the corresponding entry.
4. **Linking table(LINKTAB)** contains information about the external references and public definitions of the program P.

Binary Program

A binary program is a machine language program, which contains a set of program units P where Pi ε P. Pi has a memory address that is relocated at its link origin. Link origin is the address of the origin assigned by the linker while producing a binary program. You need to invoke a linker command for creating a binary program from a set of object modules. The syntax for the linker command is:

linker <link origin>, <object module names>,[<execution start address>]

3 types of Addresses

- Translated (Translation time) Address : Assigned by Translator
- Linked Address : Assigned by linker
- Load Address : Assigned by Loader

Origin

While compiling a program, the language translator needs to know what memory address of first memory word of the target program should have, this is known as **origin** of program.

3 types of Origin

- Translated (Translation time) Origin : Address of the origin used by the translator.
- Linked Origin : Address of the origin used by linker while producing a binary program.
- Load Origin : Address of the origin used by loader while loading the program in memory for execution.

Design of Linker

Design of Linker

The design of Linker or Linkage Editor involves two things :

- Program Relocation
- Program Linking

Program Relocation

- Program relocation is the process of modifying the addresses containing instructions of a program.
- You need to use program relocation to allocate a new memory address to the instruction. Instructions are fetched from the memory address and are followed sequentially to execute a program.
- The relocation of a program is performed by a linker and for performing relocation you need to calculate the relocation_factor that helps specify the translation time address in every instruction.
- Let the translated and linked origins of a program P be t_origin and l_origin, respectively.
- The relocation factor of a program P can be defined as:

relocation_factor = l_origin – t_origin

Program Relocation Algorithm

The algorithm that you use for program relocation is:

1. program_linked_origin:=<link origin> from linker command;
2. For each object module
3. t_origin :=translated origin of the object module;
   OM_size :=size of the object module;
4. relocation_factor :=program_linked_origin-t_origin;
5. Read the machine language program in work_area;
6. Read RELOCTAB of the object module
7. For each entry in RELOCTAB
   A. **translated_addr**: = address in the RELOCTAB entry;
   B. **address_in_work**: =address of work_area + translated_address – t_origin;
   C. add relocation_factor to the operand in the wrd with the address address_in_work_area.
   D. **program_linked_origin**: = program_linked_origin + OM_size;

Program Linking Algorithm

1. **program_linked_origin**: =<link origin> from linker command.
2. for each object module
   A. **t_origin**: =translated origin of the object module;
      OM_size :=size of the object module;
   B. **relocation_factor**: =program_linked_origin – t_origin;
   C. read the machine language program in work_area.
   D. Read LINKTAB of the object module.
   E. For each LINKTAB entry with type=PD
      name: =symbol;
      linked_address: =translated_address + relocation_factor;
      Enter (name, linked_address) in NTAB.
   F. Enter (object module name, program_linked_origin) in NTAB.
   G. **Program_linked_origin**: = program_linked_origin + OM_size;
3. for each object module
   A. **t_origin**: =translated origin of the object module;
      program_linked_origin :=load_address from NTAB;
   B. for each LINKTAB entry with type=EXT
      - **address_in_work_area**: =address of work_area + program_linked_origin - <link origin> + translated address – t_origin
      - search symbol in NTAB and copy its linked address. Add the linked address to the operand address in the word with the address address_in_work_area.

Program Linking

•Linking in a program is a process of binding an external reference to a correct link address.

•You need to perform linking to resolve external reference that helps in the execution of a program.

• All the external references in a program are maintained in a table called name table (NTAB), which contains the symbolic name for external references or an object module.

•The information specified in NTAB is derived from LINKTAB entries having type=PD.

Self Relocating Programs

•A self relocating program performs the relocation of its own address sensitive instructions.

•This way it can execute in any area of memory.

•This property is important because the operating system may assign a different memory area every time the program is to be executed.

•A self relocating program is designed to have the following 2 components in addition to machine Language instructions that perform the program‘s logic:

- ❑ A table of information concerning the address sensitive instructions .This table resembles RELOCTAB.
- ❑ Code to perform the relocation of address sensitive instructions described in the table. This code is called the relocating logic.

References

Book:-

- System Programming, John J. Donovan, Chapter 3, Page No. 111 - 117.

Queries???

Thank You

UNIVERSITY INSTITUTE of COMPUTING

Asia's Fastest Growing University

System Software
CAT-204

Design By:

Prof. Pawandeep Sharma

A.P

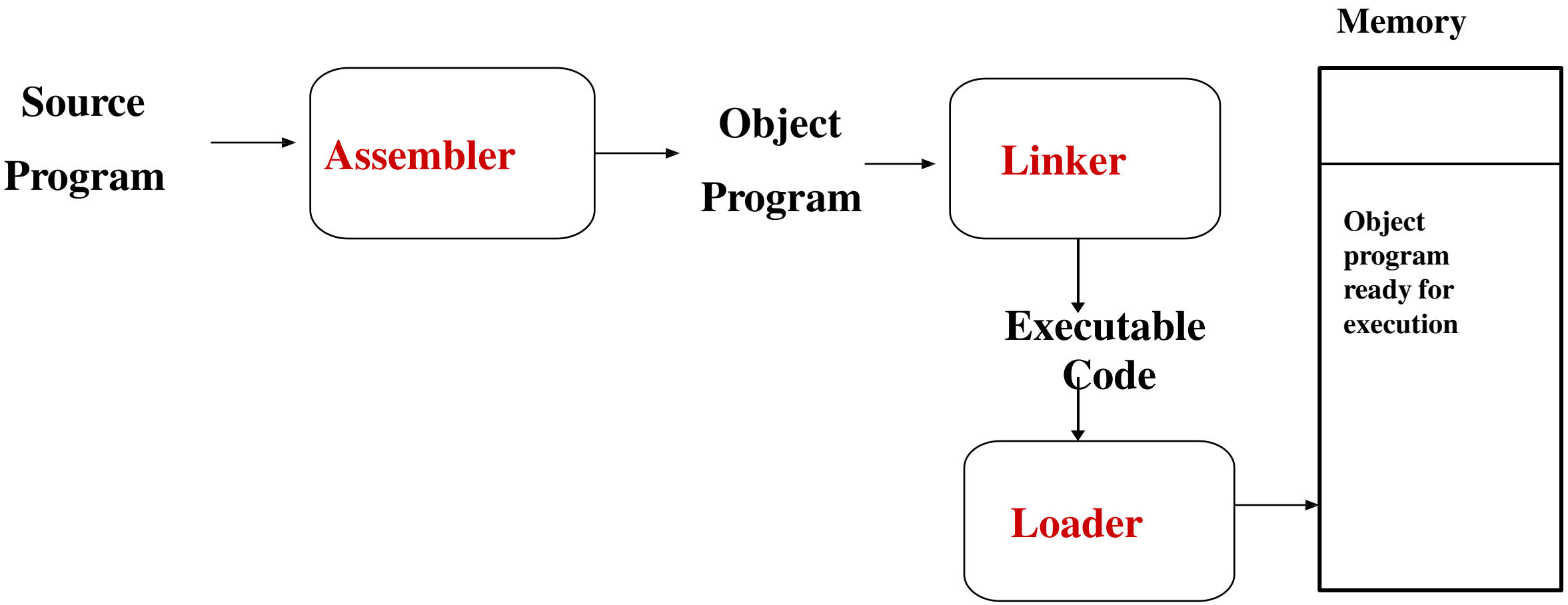
Chandigarh University-Gharuan

LOADERS

BASIC DEFINITION

- Loader is a system software program that performs the loading function.
- Loading is the process of placing the program into memory for execution.
- The loader is responsible for initializing the execution of process.

ROLE OF LOADER & LINKER



Loaders Scheme or types of Loader

1. Absolute loader

2. Relocating loader

3. Direct linking loader

Absolute Loader

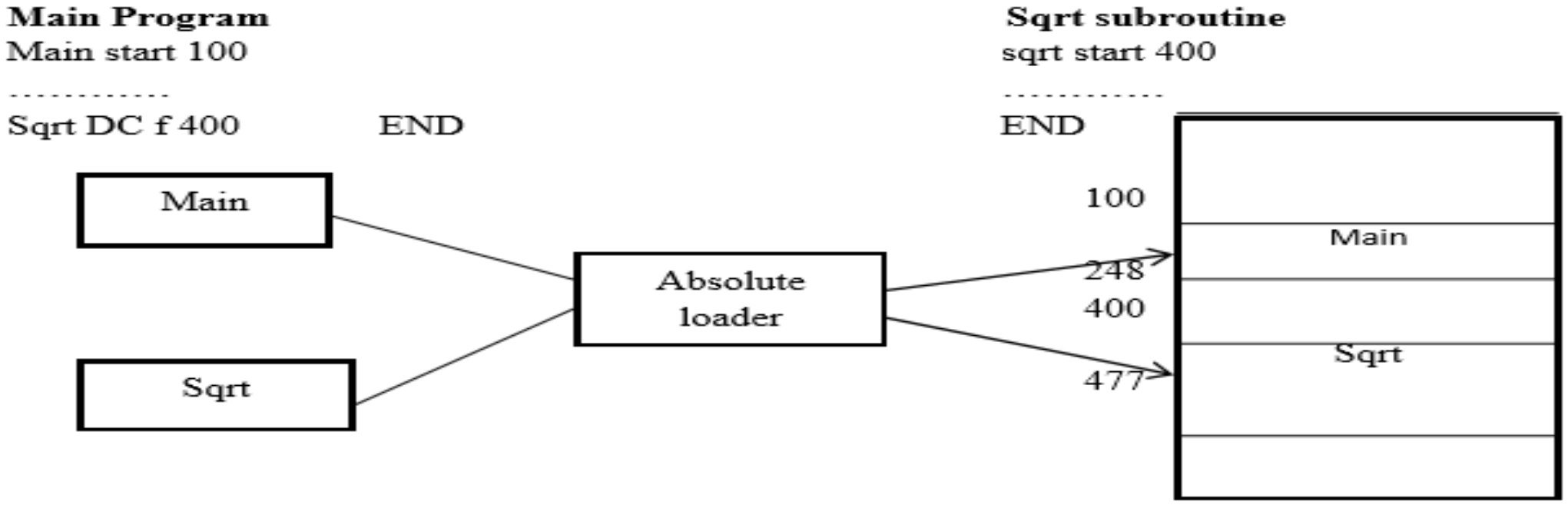
In this scheme the assembler outputs the machine language translation of the source program.

The data is punched on the cards instead of being placed directly in memory .

The loaders in turns simply accept the machine language text and places into core at the location prescribed by the assembler.

Absolute Loader

The main program assigned to location 100 to 247 and the subroutine is assigned to the location 400 to 477, if the changes were made to main memory i,e., increased its length more to an 300 byte at that time relocation is necessary



Absolute Loader

Design of an absolute loader: We can design an absolute loader we must and should having two types cards 1. Text cards. 2. Transfer cards.

1 Text cards: This type of card is used to store instructions and data. The capacity of this card is 80bytes. It must convey the machine instructions that assembler has created along with assigned core location.

Card column	content
1	Card type=0[indicates text card
2	Count the number of bytes in information
3-5	Address of that information
6-7	Empty
8-72	Instruction and data to be loaded
73-80	Cards sequence numbers

Absolute Loader

2 Transfer cards: These cards must convey the entry point of the program, which is where the loader is to transfer the control when all instructions are loaded.

Card column	content
1	Card type=1[indicates transfer card
2	Count type=0
3-5	Address of entry points
6-72	Empty
73-80	Cards sequence numbers

Absolute Loader

Algorithm for an absolute loader:

Statement 1: start

Statement 2: read header record[first record or first line]

Statement 3: program length

Statement 4: if[it is text card or transfer card]

If it is text card,then store the data and instruction

Else

Transfer instructions

Statement 5: code is in character for then it will convert in to internal representation

Statement 6: read next object program

Statement 7: end

Absolute Loader

Advantages:

1. It is very simple and easy to implement.
2. Multiple segments can be allowed at a time

Disadvantages:

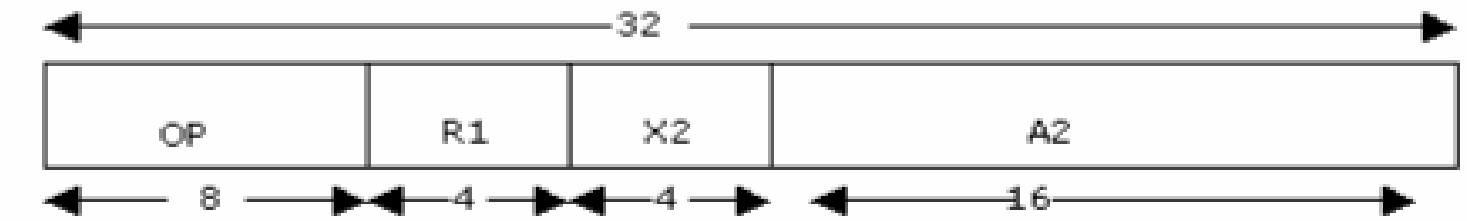
1. In this loader program adjust all internal segment addresses. So that programmers must and should know the memory management and address of the programs.
2. If any modification is done in one segment then starting address is also changed.

Relocating Loaders

- This loader assembles each procedure segment independently and passes the text and information to relocation and intersegment references.
- Relocating loaders was introduced in order to avoid possible reassembling of all subroutines when a single subroutine is changed.
- It also allows you to perform the tasks of allocation and linking for the programmer.
- The example of relocating loaders includes the Binary Symbolic Subroutine (BSS) loader.
- The BSS loader allows only one common data segment, it allows several procedure segments.

Relocating Loaders

- It make use of transfer vector for relocation.
- The output of the relocating assembler is the object program and information about all the programs to which it references.
- Additionally, it also provides relocation information for the locations that need to be changed.
- The BSS loader scheme is mostly used in computers with a fixed-length direct-address instruction format. Consider an example in which the 360 RX instructi



Relocating Loaders

Advantage of relocating loader:

1. Reassembling is not necessary
2. All the function of the loader are implemented only by the BSS loader

Disadvantage of relocating loader:

1. The transfer vector increases the size of the object program in memory.

Direct Linking Loaders

- A direct-linking loader is a general relocating loader and is the most popular loading scheme presently used.
- This scheme has an advantage that it allows the programmer to use multiple procedure and multiple data segments.
- In addition, the programmer is free to reference data or instructions that are contained in other segments.
- The direct linking loaders provide flexible intersegment referencing and accessing ability.

Direct Linking Loaders

An assembler provides the following information to the loader along with each procedure or data segment:

- Length of segment.
- List of all the symbols and their relative location in the segment that are referred by other segments.
- Information regarding the address constant which includes location in segment and description about the revising their values.
- Machine code translation of the source program and the relative addresses assigned.

Direct Linking Loaders

There are 4 types of cards available in the direct linking loader. They are:

1. ESD-External symbol dictionary
2. TXT-card
3. RLD-Relocation and linking dictionary
4. END-card

ESD card

It contains information about all symbols that are defined in the program but reference some where It contains:

- Reference number
- Symbol name
- Type Id
- Relative location
- Length

There are again ESD cards classified into 3 types of mnemonics. They are:

1. SD [Segment Definition]: It refers to the segment definition [01]
2. LD; It refers to the local definition [ENTRY] [02]
3. ER: it refers to the external reference they are used in the [EXTRN] pseudo op code [03]

TXT, RLD & END card

2 TXT Card: It contains the actual information are text which are already translated.

3 RLD Card: This card contains information about location in the program whose contexts depends on the address at which the program is placed. In this we are used „+“ and „–„sign, when we are using the „+“ sign then n0o need of relocation, when we are using „-„sign relocation is necessary.

The format of RLD contains: 1. Reference number 2. Symbol 3. Flag 4. Length 5. Relative location

4 END Card: It indicates end of the object program. Note: The size of the above 4 cards is 80 bytes

Compile & Go Loader

•It is a link editor or program loader in which the assembler its self places the assembled instruction directly into the designated memory location.

•After completion of assembly process it assigns the starting address of the program to the location counter, and then there is no stop between the compilation, link editing, loading, and execution of the program

Compile & Go Loader

Advantages: They are simple and easier to implement

Disadvantages:

- This loader can perform and take only one object program at a time .
- A portion of memory is wasted because of the memory occupied by the assembler for each object program due to that assembler necessary to retranslate the user program every time .
- It is very difficult to handle multiple segments or subprograms

Subroutine Linkages

- If one main program is transfer to sub program and that sub program also transfer to another program.
- The assembler does not know this mechanism[symbolic reference] hence it will declare the error message.
- That situation assembler provides two pseudo-op codes.
- They are 1 EXTRN 2 ENTRY
- The assembler will inform the loader that these symbols may be referred by other programs

Subroutine Linkages

1 EXTRN:
The EXTRN pseudo op code is used to maintain the reference between 2 or more subroutines.

2 ENTRY:
ENTRY pseudo-op code is used to defining entry locations of sub-routines. The assembler pseudo-op code ENTRY followed by a list of symbols indicates that these symbols are defined in present program and referenced in other program.

BINDERS

This type of loader performs the loading process into 2 separate programs:

- A core binder :The core binder performs the function of the allocation, relocation and linking

- A module loader :The module loader loads the module into memory i.e it performs the function of loading.

BINDERS

There are 2 major classes of binders:

1. Core image builder: A specific memory allocation of the program is performed at a time that the subroutines are bound together. It is called a core image module and the corresponding binder is called a core image builder
2. Linkage editor :The linkage editor can keep track of relocation information so that the resulting load module can be further relocated and loaded.

References

BOOKS:-

•System Programming, Dhamdhare, Chapter 3.

•https://www.youtube.com/watch?v=VG9VopzV_T0

• http://whatis.techtarget.com/definition/system-software

• http://searchdatacenter.techtarget.com/definition/assembler

•http://www.icse.s5.com/notes/m2.html

Queries???

UNIVERSITY INSTITUTE *of*
COMPUTING
Asia's Fastest Growing University

Thank You

UNIVERSITY INSTITUTE of COMPUTING

Asia's Fastest Growing University

System Software
CAT-204

Design By:

Prof. Pawandeep Sharma

A.P

Chandigarh University-Gharuan

COMPILERS

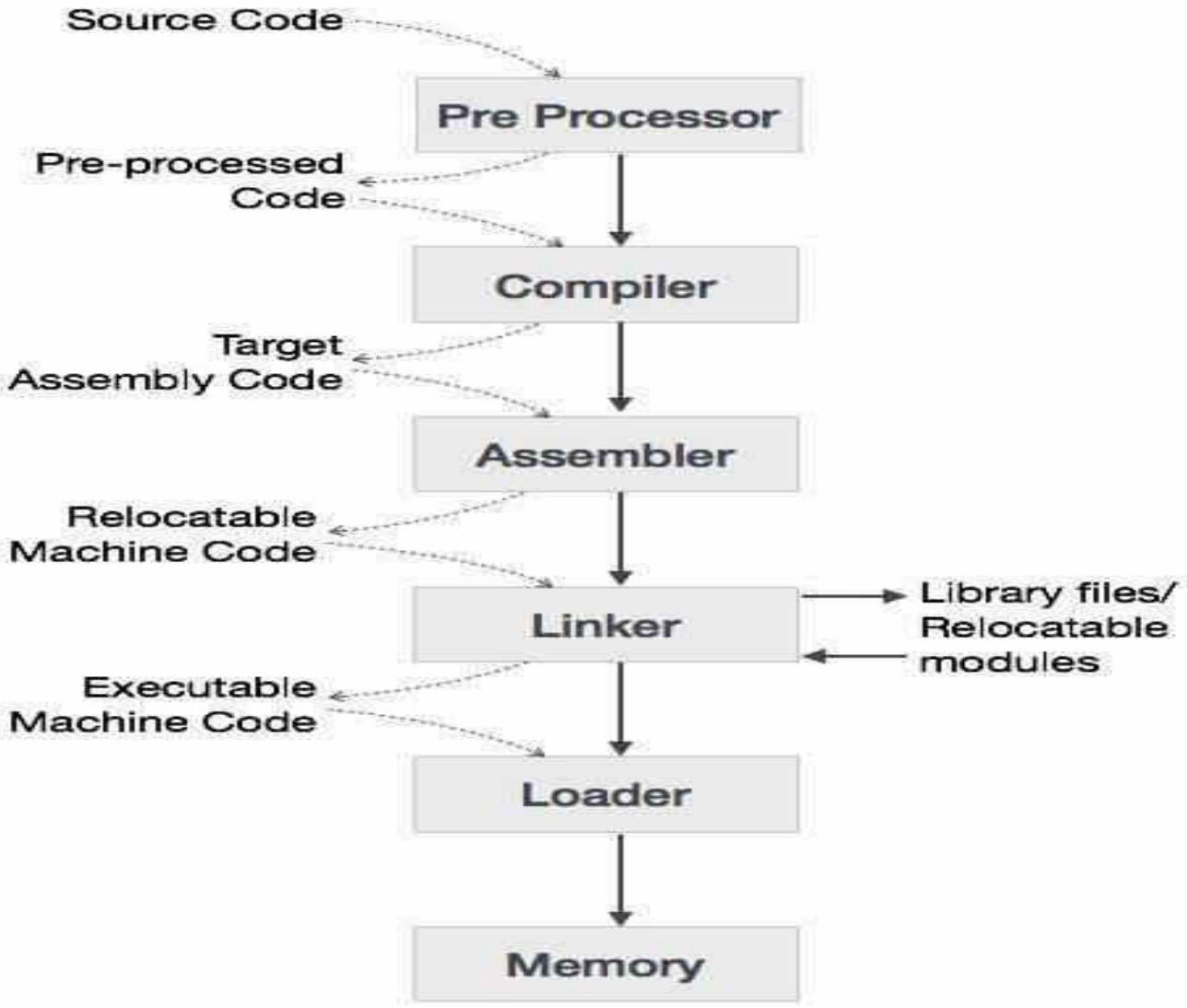
BASIC DEFINITION

•A compiler translates the code written in one language to some other language without changing the meaning of the program.

• It is also expected that a compiler should make the target code efficient and optimized in terms of time and space.

•In very simple terms - A **compiler** is a program that converts high-level language to assembly language.

The Language Processing System



COMPILER ARCHITECTURE

A compiler can broadly be divided into two phases based on the way they compile. Analysis Phase Known as the front-end of the compiler, the analysis phase of the compiler reads the source program, divides it into core parts, and then checks for lexical, grammar, and syntax errors. The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.

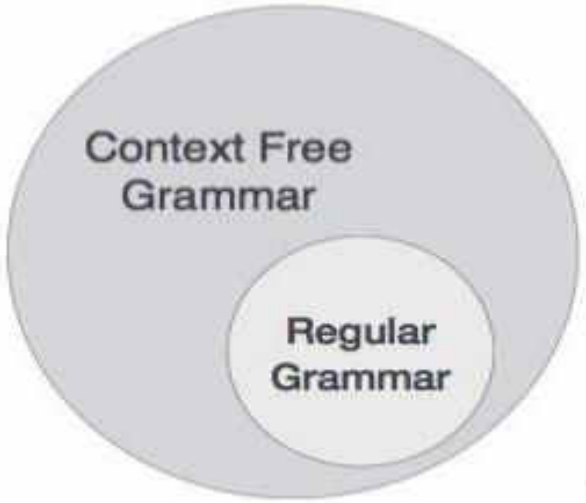
PARSING

PARSING

Syntax analysis or parsing is the second phase of a compiler.

The lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions.

Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata. CFG, on the other hand, is a superset of Regular Grammar, as depicted below:



Context-Free Grammar

A context-free grammar has four components:

A set of non-terminals (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.

A set of tokens, known as terminal symbols (Σ). Terminals are the basic symbols from which strings are formed.

A set of productions (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal called the left side of the production, an arrow, and a sequence of tokens and/or on- terminals, called the right side of the production.

One of the non-terminals is designated as the start symbol (S); from where the production begins.

TYPES OF PARSING

Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types :

- Top down parsing
- Bottom-up parsing

Top-down Parsing

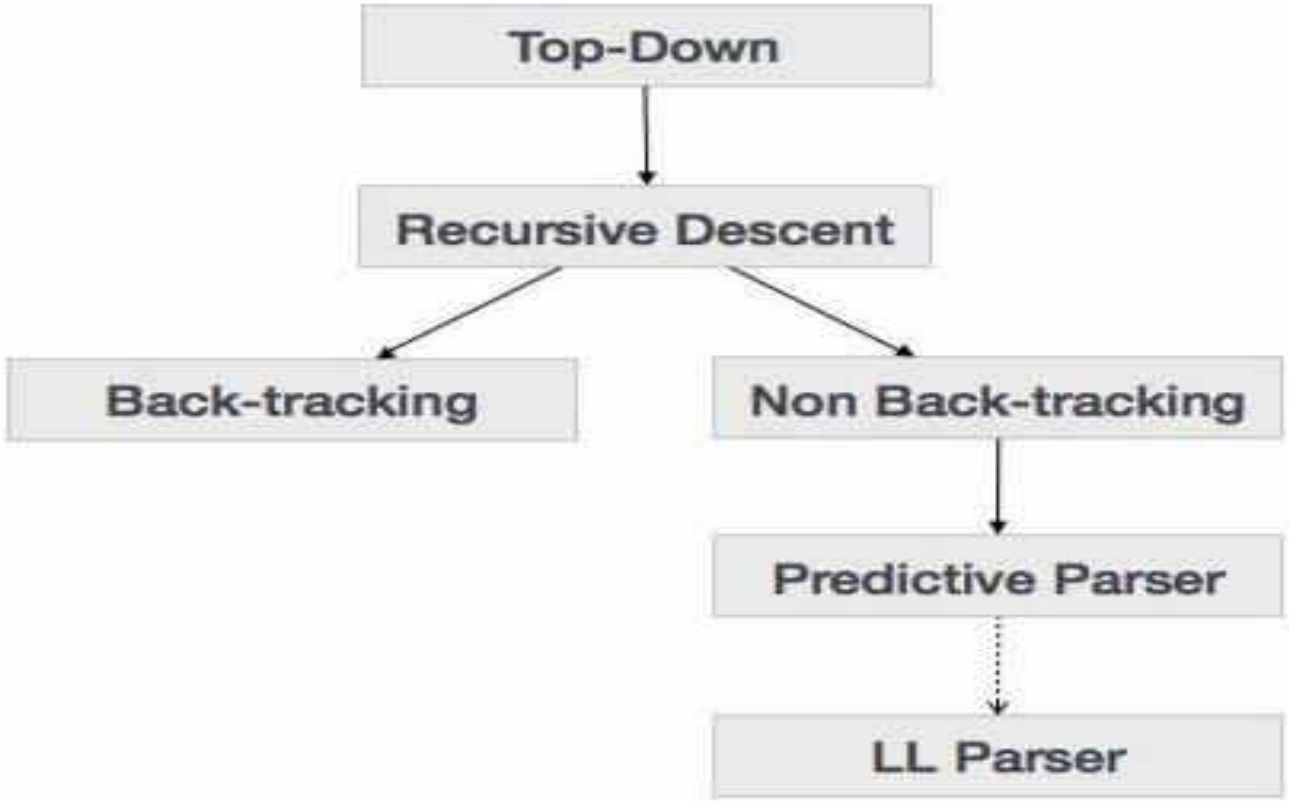
When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

Recursive descent parsing : It is a common form of top-down parsing. It is called recursive, as it uses recursive procedures to process the input. Recursive descent parsing suffers from backtracking.

Backtracking : It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production.

Top-down Parsing

The top-down parsing technique parses the input, and starts constructing a parse tree from the root node gradually moving down to the leaf nodes. The types of top-down parsing are depicted below:



Recursive Descent Parsing

- Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right.
- It uses procedures for every terminal and non terminal entity.
- This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking.
- But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing.
- This parsing technique is regarded recursive, as it uses context-free grammar which is recursive in nature.

Predictive Parser

•Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string.

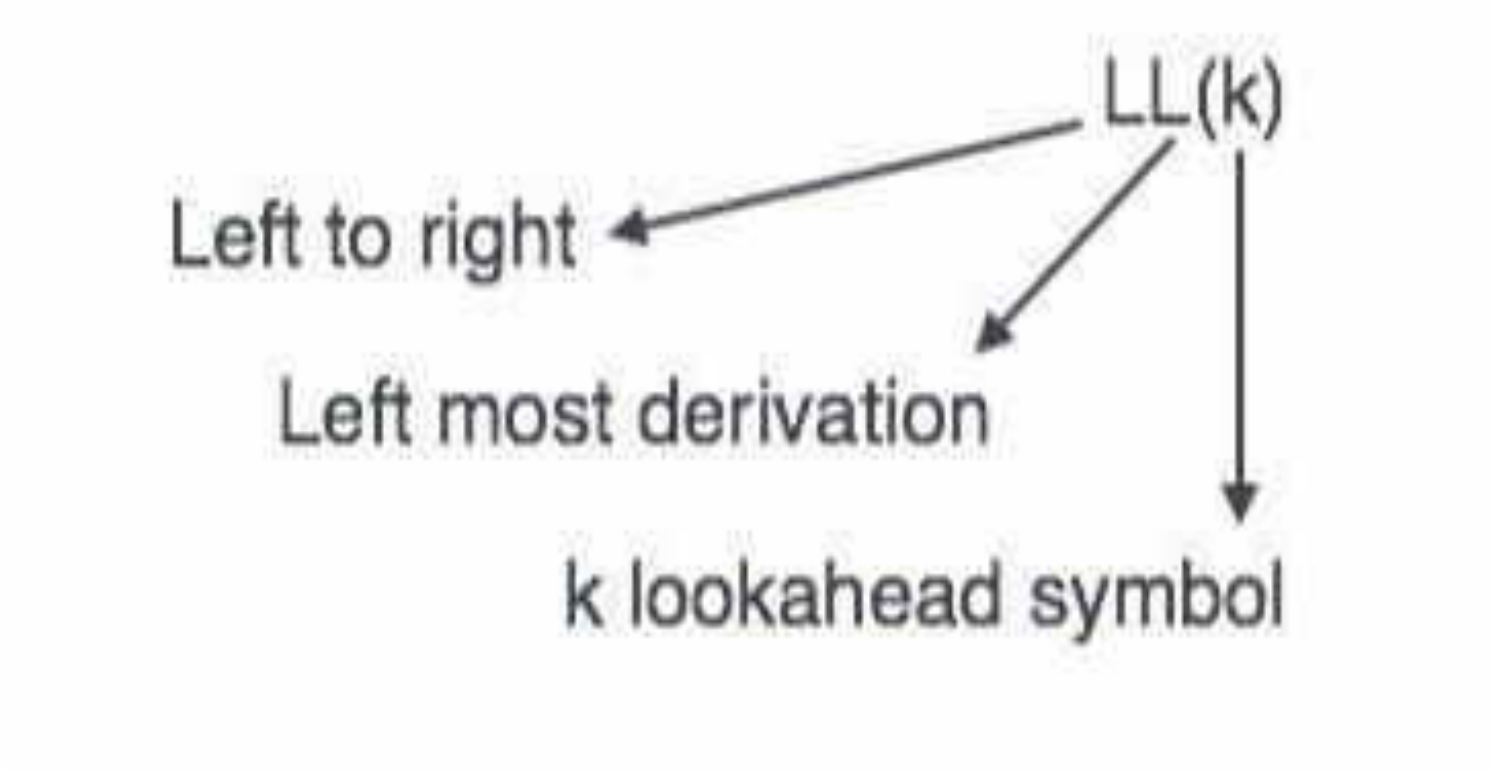
•The predictive parser does not suffer from backtracking. To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols.

• To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.

LL Parser

- An LL Parser accepts LL grammar.
- LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation.
- LL grammar can be implemented by means of both algorithms, namely, recursive-descent or table-driven. LL parser is denoted as LL(k).
- The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look aheads.
- Generally k = 1, so LL(k) may also be written as LL(1).

LL Parser



Bottom-up Parsing

Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node.

Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol.

LR Parser

- The LR parser is a non-recursive, shift-reduce, bottom-up parser.
- It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique.
- LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.

LR Parser

There are three widely used algorithms available for constructing an LR parser:

SLR(1) – Simple LR Parser:

- Works on smallest class of grammar
- Few number of states, hence very small table
- Simple and fast construction

LR(1) – LR Parser:

- Works on complete set of LR(1) Grammar
- Generates large table and large number of states
- Slow construction

LALR(1) – Look-Ahead LR Parser:

- Works on intermediate size of grammar
- Number of states are same as in SLR(1)

SYMBOL TABLE

SYMBOL TABLE

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc.

A symbol table may serve the following purposes depending upon the language in hand:

- To store the names of all entities
- To verify if a variable has been declared.
- To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- To determine the scope of a name (scope resolution).

SYMBOL TABLE

•A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:

•<symbol name, type, attribute>

•For example, if a symbol table has to store information about the following variable declaration:

•static int interest;

then it should store the entry such as:

•<interest, int, static>

Operations on SYMBOL TABLE

- A symbol table should support following operations :

- Insert()
- Lookup()

Types of SYMBOL TABLE

A compiler maintains two types of symbol tables:

- A global symbol table which can be accessed by all the procedures
- A scope symbol tables that are created for each scope in the program.

References

BOOKS:-

•System Programming, Dhamdhare, Chapter 3.

•https://www.youtube.com/watch?v=VG9VopzV_T0

• http://whatis.techtarget.com/definition/system-software

• http://searchdatacenter.techtarget.com/definition/assembler

•http://www.icse.s5.com/notes/m2.html

Queries???

UNIVERSITY INSTITUTE *of*
COMPUTING
Asia's Fastest Growing University

Thank You