MSCCS 1.3

M.Sc. (C. S.) PART I

SEMESTER - I
REVISED SYLLABUS AS PER NEP 2020

PRINCIPLES OF COMPILER DESIGN

Prof. Ravindra Kulkarni
Vice Chancellor
University of Mumbai, Mumbai

Prin. (Dr.) Ajay Bhamare
Pro Vice-Chancellor,
University of Mumbai

Professor Shivaji Sargar
Director,
CDOE, University of Mumbai

Programme Co-ordinator : **Shri Mandar Bhanushe**
Head, Faculty of Science and Technology CDOE,
Univeristy of Mumbai - 400098

Course Co-ordinator : **Sumedh Shejole**
Assistant Professor,
CDOE,Univeristy of Mumbai,
Vidyanagari, Mumbai - 400 098

Editor : **Sumedh Shejole**
Assistant Professor,
CDOE,Univeristy of Mumbai,
Vidyanagari, Mumbai - 400 098

Course Writers : **Prof. Prachi Abhijeet Surve**
Assistant Professor,
Hindi Vidya Prachar Samiti's Ramniranjan Jhunjhunwala College Of Arts, Science & Commerce, Ghatkopar West, Mumbai.

: **Sunil Vijay Palakaparambil**
Assistant Professor,
Model College of Science and commerce,
Kalyan East.

: **Mrs. Pratibha Prashant Londhe**
Assistant Professor,
MKSSS, BCA College of Women,
Shirgaon Ratnagiri.

: **Dr Amol joglekar**
Assistant Professor,
Shri SVKM's Mithibai College of Arts
Vile Parle (W), Mumbai.

: **Shri Abhijeet Chandrakant Pawaskar**
Assistant Professor,
Atharva College Of Engineering
Malad West, Mumbai

August 2024, Print - 1

Published by : Director,
Centre for Distance and Online Education,
University of Mumbai, Vidyanagari,Mumbai - 400 098.

DTP composed and Printed by:
Mumbai University Press, Vidyanagari, Santacruz (E), Mumbai- 400098

CONTENTS

Chapter No.	Title	Page No.
1.	Introduction to Compiler Design	01
2.	Introduction to Lexical Analysis	23
3.	Syntax Analysis.	55
4.	Semantic Analysis	73
5.	Intermediate Code Generation	93
6.	Code Optimization	107
7.	Runtime Environments	123
8.	Introduction to Compiler Tools	151

M.SC. (C. S.) PART I

SEMESTER - I

(PRINCIPLES OF COMPILER DESIGN)

SYLLABUS

Programme Name: M.Sc. Computer Science (Semester I) **Total Credits:** 02 **College assessment:** 25	**Course Name:** Principles of Compiler Design **Total Marks:** 50 **University assessment:** 25

Prerequisite: Programming Language concepts, Data Structures and Algorithms, Discrete Mathematics.

Course outcomes:

- Understand the theoretical foundations and concepts underlying the design and implementation of compilers.
- Acquire knowledge about the different phases of the compilation process
- Learn how to design and implement lexical analyzers and parsers
- Gain hands-on experience in building semantic analyzers
- Understand intermediate code generation and Implement optimization techniques
- Gain practical experience in code generation
- Familiarity with runtime environments and Develop skills in error handling and debugging
- Explore advanced topics in compiler design and Apply knowledge to practical projects

Course Code	Course Title	Total Credits
PSCS505	**Principles of Compiler Design**	**02**
MODULE - I **Unit 1: Front end of Compiler** **Introduction to Compiler Design**: Role and importance of compilers, Phases of compilation process, Compiler architecture and components **Lexical Analysis**: Role of lexical analyzer, Regular expressions and finite automata, Lexical analyzer generators (e.g., Lex) **Syntax Analysis**: Role of parser, Context-free grammars, Top-down parsing (LL parsing) Bottom-up parsing (LR parsing), Syntax analyzer generators (e.g., Yacc/Bison) **Semantic Analysis**: Role of semantic analyzer, Symbol table management, Type checking and type systems, Attribute grammars **Intermediate Code Generation**: Intermediate representations (IR), Three-address code generation, Quadruples and triples, Syntax-directed translation **Unit 2: Back end of Compiler** **Code Optimization**: Data flow analysis, Common subexpression elimination, Constant folding and propagation, Loop optimization techniques **Code Generation**: Code generation techniques, Target machine description, Register allocation, Instruction selection and scheduling **Runtime Environments**: Activation records and stack management. Heap memory management, Call and return mechanisms, Exception handling Lexical and Syntax Error Handling: Error recovery strategies Error reporting and handling		**02**

Introduction to Compiler Tools, Techniques and Advanced Topics in Compiler Design: Lexical and syntax analyzer generators, Code generation frameworks (e.g., LLVM), Debugging and testing compilers, Just-in-time (JIT) compilation, Parallel and concurrent programming support, Compiler optimization frameworks, Domain-specific language (DSL) compilation	

Text Books:

1. Compilers: Principles, Techniques, and Tools" by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman 2nd Edition, Pearson Publication, 2006 ISBN-13: 978-0321486813

Reference Books:

1. Modern Compiler Implementation in C" by Andrew W. Appel, 3rd Edition, Cambridge University Press, 2020, ISBN-13: 978-1108426631
2. Principles of Compiler Design" by D. M. Dhamdhere, 2nd Edition Publisher: McGraw-Hill Education, 2017, ISBN-13: 978-9339204608

1

INTRODUCTION TO COMPILER DESIGN

Unit Structure

1.0 Objective

1.1 Front end of Compiler

1.2 Introduction to Compiler Design:

1.3 Role and importance of compilers

1.4 Phases of compilation process

1.5 Compiler architecture and components

1.6 Summary

1.7 Exercise

1.8 References

1.0 OBJECTIVE

This objective of this chapter is :

- To introduce the compiler.
- To give a high level overview of the structure of a typical compiler, and discuss the trends in programming languages and machine architecture that are shaping compilers.
- To include some observations on the relationship between compiler design and computer-science theory and an outline of the applications of compiler technology that go beyond compilation.
- To give a brief outline of key programming-language concepts that will be needed for our study of compilers.

1.1 FRONT END OF COMPILER

All of these phases of a general Compiler are conceptually divided into The Front-end, and The Back-end. This division is due to their dependence on either the Source Language or the Target machine. **This model is called an Analysis & Synthesis model of a compiler.**

The **Front-end of the compiler** consists of phases that depend primarily on the Source language and are largely independent on the target machine. For example, the front-end of the compiler includes Scanner, Parser, Creation of Symbol table, Semantic Analyzer, and the Intermediate Code Generator.

The **Back-end of the compiler** consists of phases that depend on the target machine, and those portions don‘t depend on the Source language, just the Intermediate language. In this we have different aspects of the Code Optimization phase, code generation along with the necessary Error

handling, and Symbol table operations.

- The front end consists of those phases that depend primarily on source language and largely independent of the target machine.
- It includes lexical analysis, syntax analysis, semantic analysis, intermediate code generation and creation of symbol tables.
- Certain amount of code optimization can be done by the front end.
- It includes following phases:
    - **Lexical analysis**
        - The lexical analyzer is the first phase of the compiler.
        - Its Main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.
        - It is implemented by making the lexical analyzer be a subroutine.
        - Upon receiving a "get next token" command from parser, the lexical analyzer reads the input character until it can identify the next token.
        - It may also perform secondary tasks at the user interface.
        - One such task is stripping out from the source program comments and white space in the form of blanks, tabs, and newline characters.
        - The scanner is responsible for doing simple task while lexical analysis does the more complex task
    - **Syntax analysis**
        - Syntax analysis is also called hierarchical analysis or parsing.
        - The syntax analyzer checks each line of the code and spots every tiny mistake that the programmer has committed while typing the code.
        - If code is error free then syntax analyzer generates the tree
    - **Semantic analysis**
        - Semantic analyzer determines the meaning of a source string.
        - For example matching of parentheses in the expression, or matching of if..else statement or performing arithmetic operation that are type compatible, or checking the scope of operation
    - **Intermediate code generation**
        - The intermediate representation should have two important properties, it should be easy to produce and easy to translate into a target program.

- We consider an intermediate form called “three address code”.
- Three address codes consist of a sequence of instructions, each of which has at most three operands.

- **Creation of symbol table**
  - A symbol table is a data structure used by a language translator such as a compiler or interpreter.
  - It is used to store names encountered in the source program, along with the relevant attributes for those names.
  - Information about following entities
  - Variable/Identifier
  - Procedure/function
  - Keyword
  - Constant
  - Class name
  - Label name

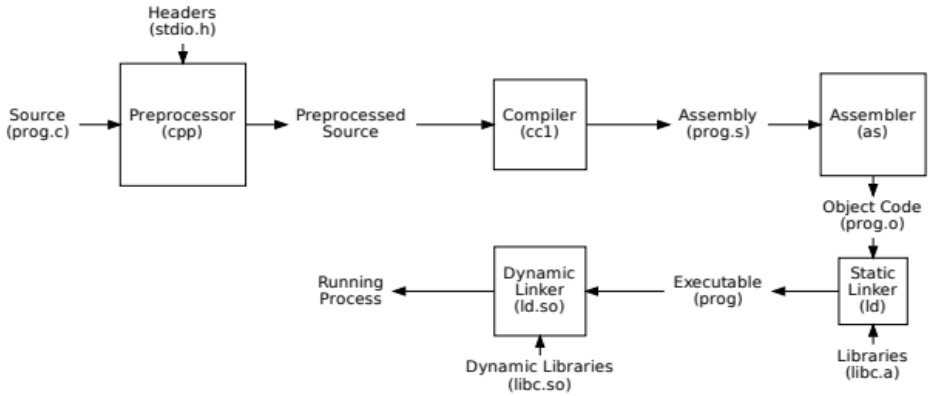
1.2 INTRODUCTION TO COMPILER DESIGN

- The software systems that do this translation are called **compilers**.
- The compiler is software that converts a program written in a high-level language also known as Source Language to a low-level language also known as Object/Target/Machine Language/0, 1’s.
- A translator or language processor is a program that translates an input program written in a programming language into an equivalent program in another language.
- The compiler is a type of translator, which takes a program written in a high-level programming language as input and translates it into an equivalent program in low-level languages such as machine language or assembly language.
- The program written in a high-level language is known as a source program, and the program converted into a low-level language is known as an object (or target) program.
- Without compilation, no program written in a high-level language can be executed. For every programming language, we have a different compiler; however, the basic tasks performed by every compiler are the same.
- The process of translating the source code into machine code involves several stages, including lexical analysis, syntax analysis,semantic analysis, code generation, and optimization.

- Compiler is an intelligent program as compared to an assembler.
- Compiler verifies all types of limits, ranges, errors , etc.
- Compiler program takes more time to run and it occupies a huge amount of memory space.
- The speed of the compiler is slower than other system software.
- It takes time because it enters through the program and then does translation of the full program.
- When the compiler runs on the same machine and produces machine code for the same machine on which it is running. Then it is called a **self compiler** or resident compiler.
- Compiler may run on one machine and produce the machine codes for other computer then in that case it is called a **cross compiler**.

The Compiler Toolchain:

- A compiler is one component in a toolchain of programs used to create executables from source code. Typically, when you invoke a single command to compile a program, a whole sequence of programs are invoked in the background.
- Following Figure shows **A Typical Compiler Toolchain** the programs typically used in a Unix system for compiling C source code to assembly code.



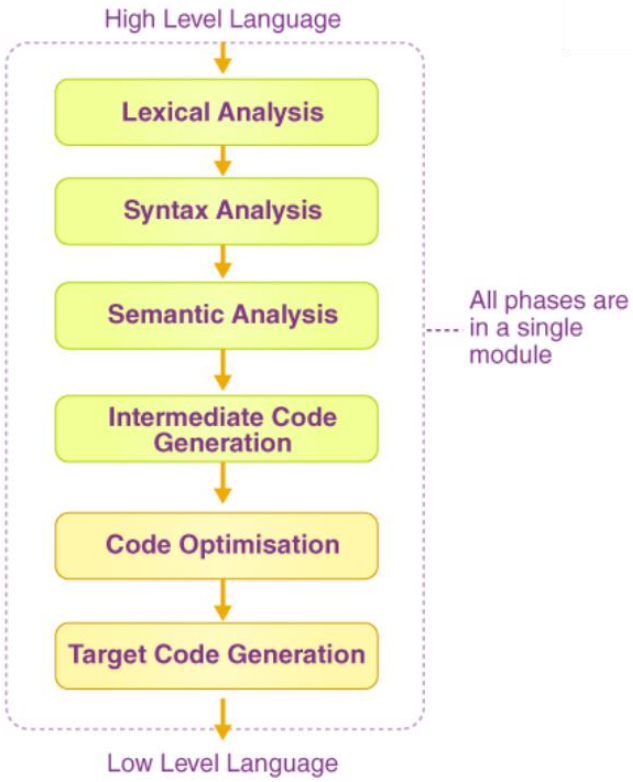
- **The preprocessor**
  - It prepares the source code for the compiler proper.
  - In the C and C++ languages, this means consuming all directives that start with the # symbol.
  - For example, an #include directive causes the preprocessor to open the named file and insert its contents into the source code.
  - A #define directive causes the preprocessor to substitute a value wherever a macro name is encountered. (Not all languages rely on a preprocessor.)

- **The compiler**
  - It properly consumes the clean output of the preprocessor.
  - It scans and parses the source code, performs type checking and other semantic routines, optimizes the code, and then produces assembly language as the output.
- **The assembler**
  - It consumes the assembly code and produces object code.
  - Object code is "almost executable" in that it contains raw machine language instructions in the form needed by the CPU.
  - However, object code does not know the final memory addresses in which it will be loaded, and so it contains gaps that must be filled in by the linker.
- **The linker**
  - It consumes one or more object files and library files and combines them into a complete, executable program.
  - It selects the final memory locations where each piece of code and data will be loaded, and then "links" them together by writing in the missing address information.
  - For example, an object file that calls the printf function does not initially know the address of the function.
  - An empty (zero) address will be left where the address must be used.
  - Once the linker selects the memory location of printf, it must go back and write in the address at every place where printf is called.

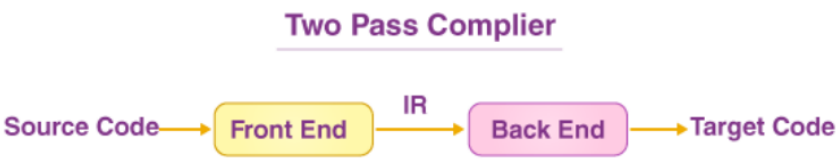
Types of Compiler

- The following are the different types of compilers that are used:
  - Single Pass Compilers
  - Two Pass Compilers
  - Multipass Compilers
  - Just-in-time (JIT) compiler
  - Cross compiler
  - Bytecode compiler
  - Source-to-source compiler
  - Binary compiler:
  - Hardware compiler

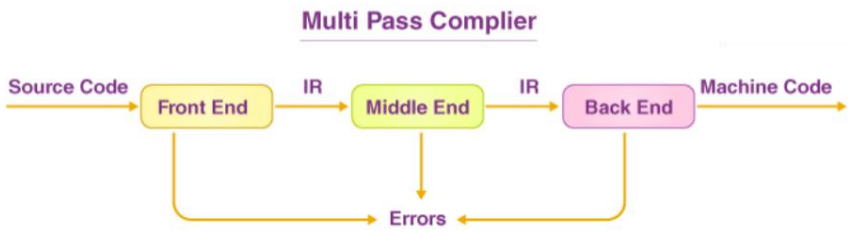
- **Single Pass Compiler**
  - When all the phases of the compiler are present inside a single module, it is simply called a single-pass compiler.
  - It performs the work of converting source code to machine code.
  - In a single-pass compiler, when a line source is processed it is scanned and the tokens are extracted.
  - Thus the syntax of the line is inspected and the tree structure and some tables including data about each token are constructed.
  - Finally, after the semantic element is tested for correctness, the code is created. The same process is repeated for each line of code until the whole program is compiled.
  - Usually, the entire compiler is built around the parser, which will call procedures that will perform different functions.



- **Two Pass Compiler**
  - Two-pass compiler is a compiler in which the program is translated twice, once from the front end and the back from the back end known as Two Pass Compiler.



- **Multipass Compiler**
  - When several intermediate codes are created in a program and a syntax tree is processed many times, it is called a Multipass Compiler.
  - It breaks codes into smaller programs.



- **Just-in-time (JIT) compiler**
  - It compiles programs as they are executed. It is faster than traditional compilers and helps in reducing program size by elimination of redundant code.
  - This reduced the size of the program and make it more efficient.
  - This helps in performance improvement.
- **Cross compiler**
  - This is a technology to allow developers to compile and run codes on various platforms.
  - This type is useful while working on several versions of code for ensuring that all platforms are being supported.
  - This is useful while working on a new platform to verify whether the code is working on this platform.
- **Bytecode compiler**
  - It translates high-level language into machine code which is executable on the target machine.
  - Such compilers allow developers to write codes in a high-level language and compile them into machine code.
  - Through this compiler, developers write concise and comprehensible codes. These compilers should be written in high-level language.
  - They are not suitable for developing low-level code.
- **Source-to-source compiler**
  - This software tool translates the source code into executable code. Such compilers are used to translate source code written in multiple programming languages.
  - The translation process can be completed in both manual and automatic methods.

- Compilers translate source code into machine code which is executed by a target machine.

- **Binary compiler**
  - This compiler translates the source code file into binary format.
  - This type of format stores the program information in a compact form that is easily read by computer.
  - Developers use compilers for network programming, database administration, and web development.

- **Hardware compiler**
  - Such compilers compile the source code into machine code for transforming source code into machine code.
  - Post that, the computer executes this code.
  - Such compilers are used in operating systems, embedded systems, and computer games.
  - Assembler is a type of hardware compiler.

1.3 ROLE AND IMPORTANCE OF COMPILERS

ADVANTAGES OF COMPILER:

- **Improved performance:**
  - Compiled code tends to run faster than interpreted code because it has been translated into machine code that can be directly executed by the computer's processor.
  - This can be particularly important for performance-critical applications, such as scientific simulations or real-time systems.

- **Portability:**
  - Compilers allow programmers to write code in a high-level programming language that can be easily translated into machine code for a variety of different platforms.
  - This makes it easier to develop software that can run on different systems without requiring significant changes to the source code.

- **Increased Security:**
  - Compilers can help improve the security of software by performing a number of checks on the source code, such as checking for syntax errors and enforcing type safety.
  - This can help prevent certain types of vulnerabilities, such as buffer overflows and type coercion attacks.

- **Debugging support:**
  - Most compilers include a number of debugging tools that can help programmers find and fix errors in their code.
  - These tools can include features such as syntax highlighting, error messages, and debuggers that allow programmers to step through their code line by line.
- **No dependencies:**
  - Your client or anyone else doesn't need any compiler, interpreter, or third party program to be installed in their system, for executing the shared executable file of your source code.
- Compared to machine language, the notation used by programming languages is closer to the way humans think about problems.
- The compiler can spot some obvious programming mistakes.
- Programs written in a high-level language tend to be shorter than equivalent programs written in machine language.
- Another advantage of using a high- level language is that the same program can be compiled to many different machine languages and, hence, be brought to run on many different machines.
- Compilers offer a number of advantages for software development, including improved performance, portability, increased security, and debugging support.

DISADVANTAGES OF COMPILER:

- **Compilation time:**
  - Depending on the size and complexity of the source code, compilation can take a significant amount of time.
  - This can be a hindrance to productivity if frequent updates to the code are required.
- **Error detection:**
  - Compilers can only detect syntax errors and certain semantic errors, and may not catch all errors in the source code.
  - This means that the compiled program may not behave as expected, and debugging may be required to identify and fix the errors.
- **Portability:**
  - Programs compiled for a specific platform or architecture may not be able to run on other platforms or architectures without being recompiled.
  - This can be a limitation if the program needs to be run on multiple platforms.

- **Execution speed:**
  - Programs compiled from high-level languages may not be as fast as programs written in low-level languages, as the compiled code may include additional instructions for the compiler to interpret.
- **Lack of flexibility:**
  - Compilers can limit the flexibility of programs since changes often require recompilation.
- **Resource consumption:**
  - Compilers can consume system resources, particularly during the compilation process, which may affect other tasks on the machine.
- Compilers can be useful tools in software development, but they may not be suitable for all situations and may require additional effort to ensure that the compiled code is correct and efficient.

USES OF COMPILER:

- **Ease of programming:**
  - High-level programming languages are easier for humans to read and write than machine code, which is a series of numbers and symbols that can be difficult for humans to understand.
  - By using a compiler to translate high-level language into machine code, programmers can write code more quickly and easily.
- **Portability:**
  - Compilers allow programmers to write code that can be easily compiled and run on a wide variety of devices and platforms.
  - This is because the source code is independent of the underlying hardware and is only translated into machine code when it is compiled.
- **Abstraction:**
  - Compilers provide a level of abstraction between the programmer and the underlying hardware, allowing programmers to focus on the logic of their programs without having to worry about the specific details of the hardware.
- **Performance:**
  - Compilers can optimize the machine code generated from the source code, resulting in faster and more efficient programs.
- Compilers are an **essential tool** in software development, as they allow programmers to write code that is easier to read and write, can be easily compiled and run on different devices and platforms, and can be optimized for performance.

- A compiler is a program that **translates source code** written in a programming language into machine code that can be executed by a computer.
- The source code is written by a programmer in a high-level programming language, such as C++ or Java, which is easier for humans to read and write.
- The compiler **converts the source code into machine code,** which is a low-level language that can be understood and executed by the computer's processor.
- There are many different types of compilers, including ones for general-purpose programming languages and ones for specialized languages used in specific fields, such as system programming or database programming.
- They also provide a level of abstraction between the programmer and the underlying hardware, allowing programmers to focus on the logic of their programs without having to worry about the specific details of the hardware.

APPLICATIONS OF COMPILER:

- **Software development:** Compilers are an essential tool for software development because they allow programmers to write code in a high-level language that is easy to understand and debug, and then translate that code into machine code that can be efficiently executed by the computer.
- **System software:** Many operating systems, including Windows, macOS, and Linux, are written in high-level programming languages and use compilers to translate the source code into machine code.
- **Embedded systems:** Compilers are also used to develop software for embedded systems, which are small, specialized computer systems that are used in a variety of devices, such as cell phones, automobiles, and industrial control systems.
- **Scientific computing:** Compilers are used to develop software for scientific computing applications, such as simulations, data analysis, and machine learning.
- **Game development:** Compilers are used to develop software for video games, which typically require efficient performance and may be written in a variety of programming languages.
- **Embedded Systems:** Compilers are used in embedded systems development for appliances, IoT devices, and automotive control systems.
- **High-Performance Computing:** Compilers play a key role in high-performance computing clusters for scientific research and data analysis.

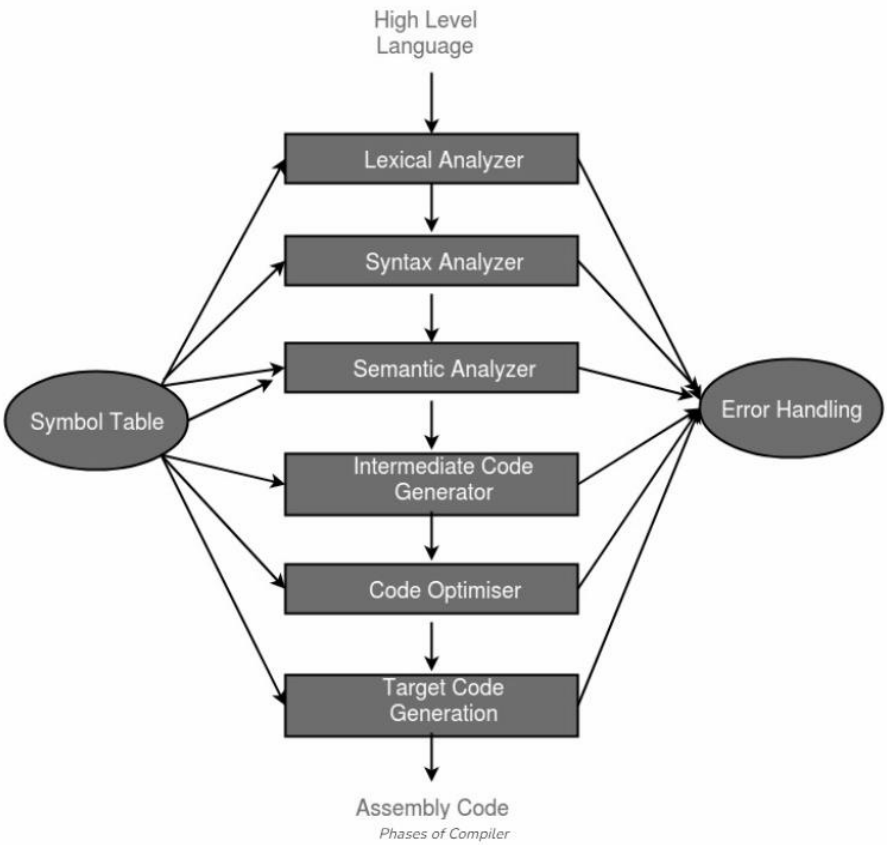
- **Utility Software:** Compilers are used to develop utility software, like text editors, database management systems, and networking tools.

Operations/Role of Compiler are as follow:

- It breaks source programs into smaller parts.
- It enables the creation of symbol tables and intermediate representations.
- It helps in code compilation and error detection.
- it saves all codes and variables.
- It analyses the full program and translates it.
- Separate compilation is supported.
- Read the full programme, analyse it, and translate it to a semantically similar language.
- Depending on the type of machine, converting source code to object code.

1.4 PHASES OF COMPILATION PROCESS

The following steps are the phases of compiler that are undertaken by it in order to convert the code to output:



Phases of Compiler

- **Lexical analysis:**
  - It is the first phase where high-level input program is converted into a sequence of tokens.
  - This can be implemented with Deterministic finite Automata.
  - The output is the sequence of tokens that are sent to the parser for syntax analysis.

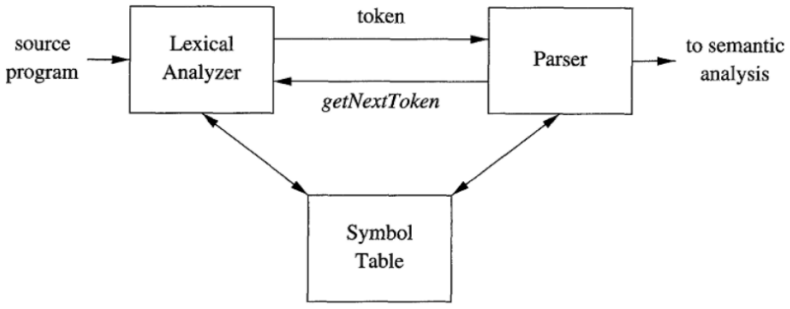


Figure : Lexical Analyzer

  - Lexical Analysis is also known as Scanning or Linear Analysis.
  - To begin, the lexical analyzer examines the entire program and divides it into tokens.
  - The string with meaning is referred to as a token.
  - The input string's class or category is described by the token.
  - Identifiers, Keywords, Constants, and so on.
  - Sentinel refers to the end of the buffer or token.
  - The token is described by a set of rules known as a pattern.
  - Lexemes are the sequence of characters in source code that correspond to the token pattern.
  - For example int, i , num etc.
  - There are two pointers in Lexical analysis they are Lexeme pointer and Forward pointer.
  - To recognize a token Regular expressions are used to construct Finite Automata.
  - Input is the source code and output is the tokens.
  - E.g.

    Input: x= x + y*z*3

    Output: Tokens or table of tokens

=	x
+	y
*	z
	3

- **Syntax Analysis/ Parsing:**
  - It is the second phase of a compiler.
  - In this phase, it verifies the syntactical structure of a given input.
  - To do so, it builds a data structure called Syntax or Parse tree.
  - The parse tree is constructed using pre-defined grammar of language and input string.
  - If a given input string can be produced using syntax tree, the input string is found to be in the correct syntax.
  - If it is not correct, the error is reported by syntax analyzer.

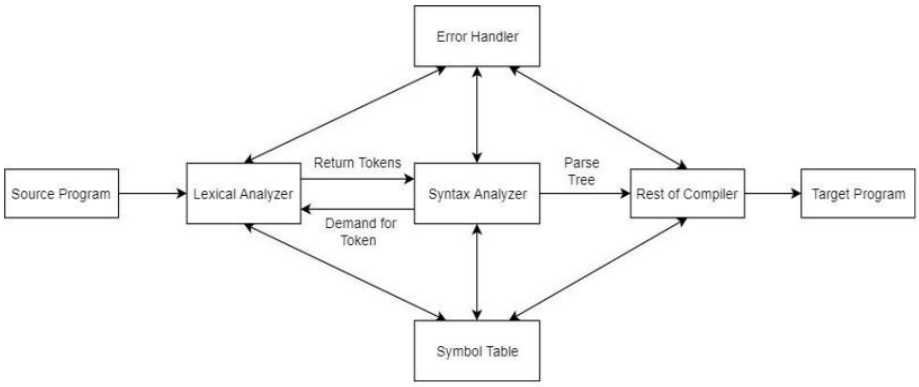


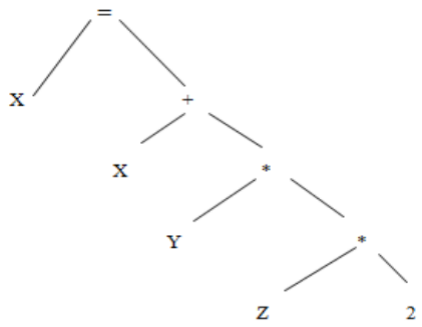
Fig : Lexical and Syntax Analyzer

- Syntax analysis, also known as syntactical analysis, parsing, or hierarchical analysis, is a type of analysis that examines the structure of a sentence.
- Syntax is the arranging of words and phrases in a language to produce well-formed sentences.
- The tokens generated by the lexical analyzer are put together to form a less detailed hierarchical structure known as the syntax tree.
- Input is token and output is syntax tree.
- Grammatical errors are checked during this phase. Example: Parenthesis missing, semicolon missing, syntax errors etc.
- For example:

Input: tokens

=	X
+	Y
*	Z
	3

Output:



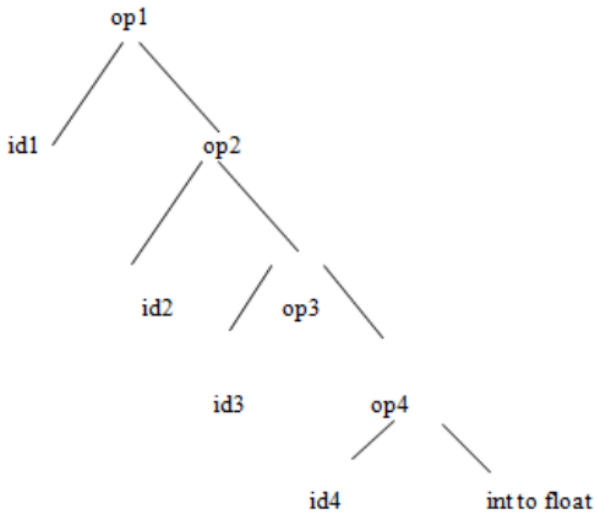
- **Semantic analysis:**
  - It is the process of interpreting meaning from text.
  - This allows the computer system to understand and interpret paragraphs, sentences and whole documents.
  - For this purpose, it analyzes the grammatical structure and identifies relationships between individual words.
  - Semantic analyzer checks the meaning of source program. Logical errors are checked during this phase. Example: divide by zero, variable undeclared etc.
  - Example of logical errors

    int a;

    float b;

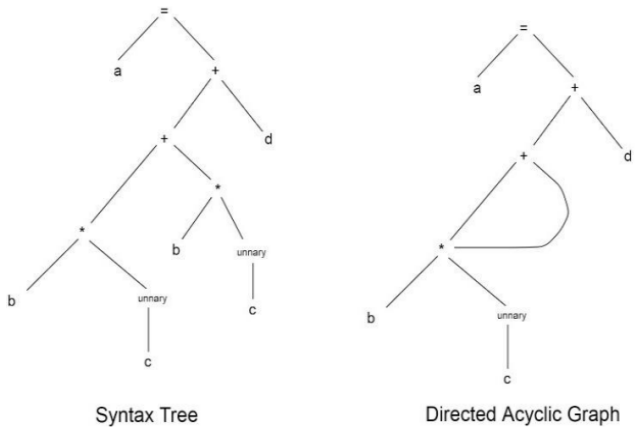
    char c;

    c=a+b;
  - Parse tree refers to the tree having meaningful data.
  - Parse tree is more specified and more detailed.
  - Input is syntax tree and output is parse tree (syntax tree with meaning) is as follows:



- **Intermediate code generation:**
  - It can translate source programs into machine program.
  - An intermediate code is generated because the compiler cannot directly generate machine code in one pass.
  - It first converts the source program into intermediate code to perform efficient generation of machine code.
  - It is represented in postfix notation, directed acyclic graph, quadruples, and triples.
  - Intermediate code (IC) is code that sits between high-level and low-level languages, or code that sits between source and target code.
  - The conversion of intermediate code to target code is simple.
  - Intermediate code functions as a bridge between the front end and the back end.
  - Three address codes, abstract syntax trees, prefix (polish), postfix (reverse polish), and other types of intermediate code exist.

Directed Acyclic Graph (DAG) is kind of abstract syntax tree which optimizes repeated expressions in syntax tree.



Syntax Tree Directed Acyclic Graph

  - The three-address code, which has no more than three operands, is the most often used intermediate code.
  - Input: Parse tree
  - Output: Three address code

    temp1=int to float(2);

    temp2=id4*t1;

    temp3=id3*t2;

    temp4=id2+t3;

    temp4=id1;

- **Code optimization:**
  - It is a program transformation technique.
  - The aim of this phase of compiler is to code improvement by enabling it to consume fewer resources and deliver high speed.
  - High-level language constructs are replaced with efficient low-level programming codes.
  - For increasing the speed of a program, unnecessary code strings are eliminated and a sequence of statements are organized.

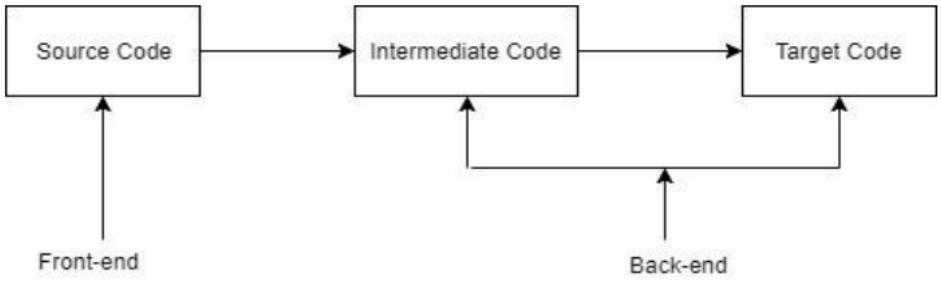


Fig : Code Optimization

  - To increase intermediate code and execution performance, code optimization is used.
  - It is vital to have code that executes faster or consumes less memory.
  - There are mainly two ways to optimize the code named Frontend (Analysis) and Back-end (Synthesis).
  - A programmer or developer can optimize the code in front-end.
  - The compiler can optimize the code on the back-end.
  - Various strategies for code optimization are listed below.
    - Compile Time Evaluation
    - Constant Folding
    - Constant Propagation
    - Common SubExpression Elimination
    - Variable Propagation
    - Code Movement
    - Loop Invariant Computation
    - Strength Reduction
    - Dead Code Elimination
    - Code Motion
    - Induction Variables and Strength Reduction.
  - Input: Three address code

- Output: : Optimized three address code

  temp1=id4*2.0;

  temp2=temp1*id3;

  id1=temp2+id2;

- **Target Code Generator:**
  - It is the final compilation phase. The generated code is an object code of lower-level programming languages such as assembly language.
  - Source code written in higher-level language is converted into a lower-level language that results in lower-level object code.
  - The main purpose of the Target Code generator is to write code that the machine can understand and also register allocation, instruction selection, etc. The output is dependent on the type of assembler. This is the final stage of compilation.
  - The optimized code is converted into relocatable machine code which then forms the input to the linker and loader.

THE GROUPING OF PHASES INTO PASSES:

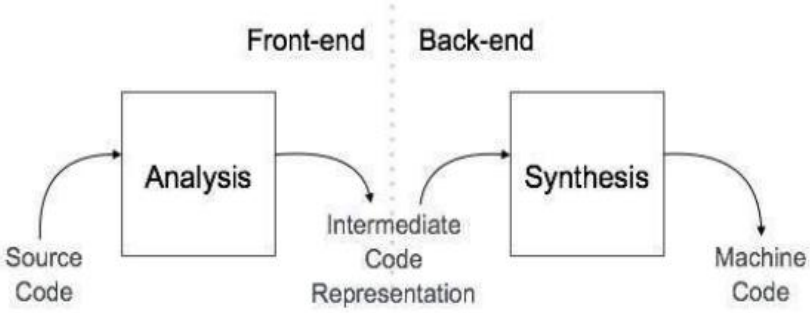
- The discussion of phases deals with the logical organization of a compiler.
- In an implementation, activities from several phases may be grouped together into a pass that reads an input file and writes an output file.
- For example, the front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass.
- Code optimization might be an optional pass.
- Then there could be a back-end pass consisting of code generation for a particular target machine.
- Some compiler collections have been created around carefully designed intermediate representations that allow the front end for a particular language to interface with the back end for a certain target machine.
- With these collections, we can produce compilers for different source languages for one target machine by combining different front ends with the back end for tha t target machine.
- Similarly, we can produce compilers for different target machines, by combining a front end with back ends for different target machines.

1.5 COMPILER ARCHITECTURE AND COMPONENTS

As we said earlier , A compiler can broadly be divided into two phases based on the way they compile as : Analysis and Synthesis phase of compiler.

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.



Synthesis Phase

- Known as the back-end of the compiler, the synthesis phase generates the target program with the help of intermediate source code representation and symbol table.
- A compiler can have many phases and passes.
- **Pass** : A pass refers to the traversal of a compiler through the entire program.
- **Phase** :
    - A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage.
    - A pass can have more than one phase.

Both analysis and synthesis are made up of internal phases.

Compiler Components:

A typical real-world compiler usually has multiple phases. This increases the compiler's portability and simplifies retargeting.

- The front end consists of the following phases:
  - **scanning:**
    - a scanner groups input characters into tokens;
    - Tokenizer (Lexical Analysis): The Tokenizer identifies and categorizes objects in each line of code, disregarding white space and comments.
    - For example, consider the line int x = 5;. The Tokenizer identifies five tokens: "int", "x", "=", "5", and ";".
  - **parsing:**
    - a parser recognizes sequences of tokens according to some grammar and generates Abstract Syntax Trees (ASTs);
  - **semantic analysis:**
    - performs type checking (ie, checking whether the variables, functions etc in the source program are used consistently with their definitions and with the language semantics) and translates ASTs into IRs;
  - **optimization:**
    - optimizes IRs.
- The back end consists of the following phases:
  - **instruction selection:**
    - maps IRs into assembly code;
  - **code optimization:**
    - optimizes the assembly code using control-flow and data-flow analyses, register allocation, etc;
  - **code emission:**
    - generates machine code from assembly code.
- The generated machine code is written in an object file.
- This file is not executable since it may refer to external symbols (such as system calls).
- The operating system provides the following utilities to execute the code:

- **linking:**
  - A linker takes several object files and libraries as input and produces one executable object file.
  - It retrieves from the input files (and puts them together in the executable object file) the code of all the referenced functions/procedures and it resolves all external references to real addresses.
  - The libraries include the operating system libraries, the language-specific libraries, and, maybe, user-created libraries.
- **loading:**
  - A loader loads an executable object file into memory, initializes the registers, heap, data, etc and starts the execution of the program.
  - Relocatable shared libraries allow effective memory use when many different applications share the same code.

1.6 SUMMARY

In this chapter we have seen basic fundamentals of compiler, like what is compiler? What are the role and importance of a compiler? Architecture and component of compiler.

1.7 EXERCISE

Answer the following:

1. Describe the various phases of compiler with suitable example
2. What is a compiler? Explain.
3. Note down the role and importance of the compiler.
4. Write a short note on components of the compiler.
5. Explain types of compiler.

1.8 REFERENCES

https://docs.google.com/document/d/1vF-JnqFttmBQph9ed61RMrrY-qBa0OBTuOph9SU1WkE/edit

https://www3.nd.edu/~dthain/compilerbook/compilerbook.pdf

Compilers: Principles, Techniques, and Tools" by Alfred V. Aho, Monica S. Lam, Ravi

Sethi, and Jeffrey D. Ullman 2nd Edition, Pearson Publication, 2006 ISBN-13: 978-0321486813

https://www.geeksforgeeks.org/advantages-and-disadvantages-of-compiler/

https://cs.lmu.edu/~ray/notes/compilerarchitecture/

❄❄❄❄❄❄❄

2

INTRODUCTION TO LEXICAL ANALYSIS

Unit Structure

2.0 Objective

2.1 Introduction to Lexical Analysis

2.2 Role of lexical analyzer

2.3 Regular expressions

2.4 Finite automata

2.5 Lexical analyzer generators (e.g., Lex)

2.6 Summary

2.7 Exercise

2.8 References

2.0 OBJECTIVE

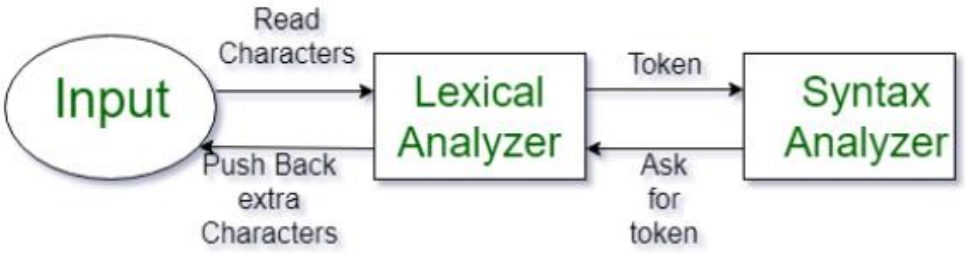
This objective of this chapter is :

- To understand how to construct a lexical analyzer.
- To implement a lexical analyzer by hand, it helps to start with a diagram or other description for the lexemes of each token.
- To identify each occurrence of each lexeme on the input and to return information about the token identified.
- To produce a lexical analyzer automatically by specifying the lexeme patterns to a lexical-analyzer generator and compiling those patterns into code that functions as a lexical analyzer.
- To show how this notation can be transformed, first into nondeterministic automata and then into deterministic automata.
- To introduce a lexical-analyzer generator called Lex (or Flex in a more recent embodiment).

2.1 INTRODUCTION TO LEXICAL ANALYSIS

- Lexical-analyzer generators by introducing regular expressions, a convenient notation for specifying lexeme patterns.
- Lexical Analysis is the first phase of the compiler also known as a scanner.
- It converts the High level input program into a sequence of Tokens.

- Lexical Analysis can be implemented with the Deterministic finite Automata.
- The output is a sequence of tokens that is sent to the parser for syntax analysis



What is Lexical Analysis?

Lexical analysis is the starting phase of the compiler.

It gathers modified source code that is written in the form of sentences from the language preprocessor. The lexical analyzer is responsible for breaking these syntaxes into a series of tokens, by removing whitespace in the source code. If the lexical analyzer gets any invalid token, it generates an error. The stream of character is read by it and it seeks the legal tokens, and then the data is passed to the syntax analyzer, when it is asked for.

There are three important terminologies used in Lexical Analysis

1. Token :

- It is a sequence of characters that represents a unit of information in the source code.
- A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.
- Example of tokens:

  Type token (id, number, real, . . .)

  Punctuation tokens (IF, void, return, . . .)

  Alphabetic tokens (keywords)

```
Keywords; Examples-for, while, if etc.
Identifier; Examples-Variable name, function name, etc.
Operators; Examples '+', '++', '-' etc.
Separators; Examples ',' ';' etc
```

- Example of Non-Tokens: Comments, preprocessor directive, macros, blanks, tabs, newline, etc.
- One token for each keyword. The pattern for a keyword is the same as the keyword itself.

- Tokens for the1 operators, either individually or in classes such as the token comparison
- One token representing all identifiers.
- One or more tokens representing constants, such as numbers and literal strings.
- Tokens for each punctuation symbol, such as left and right parentheses,comma, and semicolon.
- More examples of token:

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters **i, f**	**if**
else	characters **e, l, s, e**	**else**
comparison	< or > or <= or >= or == or !=	**<=, !=**
id	letter followed by letters and digits	**pi, score, D2**
number	any numeric constant	**3.14159, 0, 6.02e23**
literal	anything but ", surrounded by "'s	**"core dumped"**

2. Pattern: The description used by the token is known as a pattern.

Token	Lexeme	Pattern
Keyword	while	w-h-i-l-e
Relop	<	<, >, >=, <=, !=, ==
Integer	7	(0 - 9)*-> Sequence of digits with at least one digit
String	"Hi"	Characters enclosed by " "
Punctuation	,	; , . ! etc.
Identifier	number	A - Z, a - z A sequence of characters and numbers initiated by a character.

Tricky Problems When Recognizing Tokens

Usually, given the pattern describing the lexemes of a token, it is relatively simple to recognize matching lexemes when they occur on the input. However, in some languages it is not immediately apparent when we have seen an instance of a lexeme corresponding to a token. The following example is taken from Fortran, in the fixed-format still allowed in Fortran 90. In the statement

DO 5 I = 1.25

it is not apparent that the first lexeme is D05I, an instance of the identifier token, until we see the dot following the 1. Note that blanks in fixed-format Fortran are ignored (an archaic convention). Had we seen a comma instead of the dot, we would have had a do-statement

DO 5 I = 1,25

in which the first lexeme is the keyword DO.

3. Lexeme:

- A sequence of characters in the source code, as per the matching pattern of a token, is known as lexeme. It is also called the instance of a token.

- The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme.

  eg- “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;” .

Example 1 : Given C statement

printf (" Total = %d\n", s c o r e);

both printf and score are lexemes matching the pattern for token id, and "Total = °/,d\n" is a lexeme matching literal.

Example 2 :

The token names and associated attribute values for the Fortran statement

```
E = M * C ** 2
```

are written below as a sequence of pairs.

```
<id, pointer to symbol-table entry for E>
<assign_op>
<id, pointer to symbol-table entry for M>
<mult_op>
<id, pointer to symbol-table entry for C>
<exp_op>
<number, integer value 2>
```

Note that in certain pairs, especially operators, punctuation, and keywords, there is no need for an attribute value.

In this example, the token number has been given an integer-valued attribute.

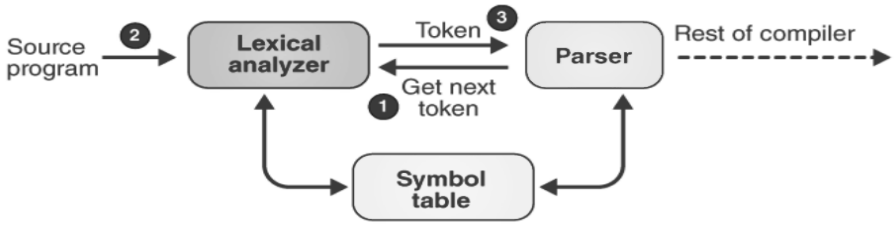
In practice, a typical compiler would instead store a character string representing the constant and use as an attribute value for number a pointer to that string.

2.2 ROLE OF LEXICAL ANALYZER

- As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program.
- The stream of tokens is sent to the parser for syntax analysis.
- It is common for the lexical analyzer to interact with the symbol table as well.
- When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.
- In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.

The lexical analyzer performs the following tasks:

- The lexical analyzer is responsible for removing the white spaces and comments from the source program.
- It corresponds to the error messages with the source program.
- It helps to identify the tokens.
- The input characters are read by the lexical analyzer from the source code.
- Stripping out comments and white spaces from the program
- Read the input program and divide it into valid tokens
- Find lexical errors
- Return the Sequence of valid tokens to the syntax analyzer
- When it finds an identifier, it has to make an entry into the symbol table.
- Figure : Interactions between the lexical analyzer and the parser



- Commonly, the interaction is implemented by having the parser call the lexical analyzer.
- The call, suggested by the **getNextToken** command, causes the lexical analyzer to read characters
- from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.
- Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes.
- One such task is stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).
- Another task is correlating error messages generated by the compiler with the source program.
- For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.

- In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions.
- If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.
- Sometimes, lexical analyzers are divided into a cascade of two processes:
    a) Scanning consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.
    b) Lexical analysis proper is the more complex portion, where the scanner produces the sequence of tokens as output.

Lexical Analysis Versus Parsing:

- There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.
- **Simplicity of design is the most important consideration.**
    - The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks.
    - For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer.
    - If we are designing a new language, separating lexical and syntactic concerns can lead to a cleaner overall language design.
- **Compiler efficiency is improved.**
    - A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing.
    - In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.
- **Compiler portability is enhanced.**
    - Input-device-specific peculiarities can be restricted to the lexical analyzer.

Advantages Of Lexical Analysis

- Lexical analysis helps the browsers to format and display a web page with the help of parsed data.
- It is responsible to create a compiled binary executable code.
- It helps to create a more efficient and specialised processor for the task.

DISADVANTAGES OF LEXICAL ANALYSIS

- It requires additional runtime overhead to generate the lexer table and construct the tokens.
- It requires much effort to debug and develop the lexer and its token description.
- Much significant time is required to read the source code and partition it into tokens.

2.3 REGULAR EXPRESSIONS

- Suppose we wanted to describe the set of valid C identifiers.
- It is almost exactly the language described as;

L(L U D)* is the set of all strings of letters and digits beginning with a letter

the only difference is that the underscore is included among the letters.

Example Let L be the set of letters $\{A, B, \ldots, Z, a, b, \ldots, z\}$ and let D be the set of digits $\{0, 1, \ldots 9\}$. We may think of L and D in two, essentially equivalent, ways. One way is that L and D are, respectively, the alphabets of uppercase and lowercase letters and of digits. The second way is that L and D are languages, all of whose strings happen to be of length one. Here are some other languages that can be constructed from languages L and D, using the operators :

1. L U D is the set of letters and digits — strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.
2. LD is the set c-f 520 strings of length two, each consisting of one letter followed by one digit.
3. L‘ is the set of all 4-letter strings.
4. L^* is the set of all strings of letters, including e, the empty string.
5. $L(L \text{ U } D)^*$ is the set of all strings of letters and digits beginning with a letter.
6. D^- is the set of all strings of one or more digits.

- In the above Example, we were able to describe identifiers by giving names to sets of letters and digits and using the language operators union, concatenation,and closure.
- This process is so useful that a notation called regular expressions has come into common use for describing all the languages that can be built from these operators applied to the symbols of some alphabet.
- In this notation, if letter- is established to stand for any letter or the underscore, and digit- is established to stand for any digit, then we

could describe the language of C identifiers by:

*letter- (letter- ¦ digit)**

- The vertical bar above means union, the parentheses are used to group subexpressions, the star means "zero or more occurrences of," and the juxtaposition of letter, with the remainder of the expression signifies concatenation.
- The regular expressions are built recursively out of smaller regular expressions, using the rules described below.
- Each regular expression r denotes a language L(r), which is also defined recursively from the languages denoted by r ' s subexpressions.
- Here are the rules that define the regular expressions over some alphabet £ and the languages that those expressions denote.

BASIS: There are two rules that form the basis:

1. e is a regular expression, and L(e) is {e}, that is, the language whose sole member is the empty string.
2. If a is a symbol in E, then a is a regular expression, and L(a) = {a}, that is, the language with one string, of length one, with a in its one position.

Note that by convention, we use italics for symbols, and boldface for their corresponding regular expression.

INDUCTION: There are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose r and s are regular expressions denoting languages L(r) and L(s), respectively.

1. (r)|(s) is a regular expression denoting the language L(r) U L(s).
2. (r)(s) is a regular expression denoting the language L(r)L(s).
3. (r)* is a regular expression denoting (L(r))*.
4. (r) is a regular expression denoting L(r). This last rule says that we can

As defined, regular expressions often contain unnecessary pairs of parentheses.

We may drop certain pairs of parentheses if we adopt the conventions that:

a) The unary operator * has highest precedence and is left associative.

b) Concatenation has second highest precedence and is left associative.

however, when talking about specific characters from the ASCII character set, we shall generally use teletype font for both the character and its regular expression.

c) | has lowest precedence and is left associative.

Under these conventions, for example, we may replace the regular expression (a)|((b)*(c)) by a|b*c. Both expressions denote the set of strings that are either a single a or are zero or more 6's followed by one c.

Example 3 :

Let £ = {a,6}.

1. The regular expression a|b denotes the language {a, b}.

2. (a|b)(a|b) denotes {aa, ah, ba, bb}, the language of all strings of length two over the alphabet E. Another regular expression for the same language is aa|ab|ba|bb.

3. a* denotes the language consisting of all strings of zero or more a's, that is, { e , a , a a , a a a , . . . }.

4. (a|b)* denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and 6's: {e,a, b,aa, ab, ba, bb,aaa,...}.

Another regular expression for the same language is (a*b*)*.

5. a|a*b denotes the language {a, b, ab, aab, aaab,...}, that is, the string a and all strings consisting of zero or more a's and ending in b.

A language that can be defined by a regular expression is called a regular set.

If two regular expressions r and s denote the same regular set, we say they are equivalent and write r = s.

For instance, (a|b) = (b|a). There are a number of algebraic laws for regular expressions; each law asserts that expressions of two different forms are equivalent.

Following Figure shows some of the algebraic laws that hold for arbitrary regular expressions r, s, and t.

LAW	DESCRIPTION
$r\|s = s\|r$	| is commutative
$r\|(s\|t) = (r\|s)\|t$	| is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt; \quad (s\|t)r = sr\|tr$	Concatenation distributes over |
$er = re = r$	e is the identity for concatenation
$r^* = (r\|e)^*$	e is guaranteed in a closure
	* is idempotent

Regular Definitions:

For notational convenience, we may wish to give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols.

If £ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

$$d_i \rightarrow r_i$$
$$d_2 \rightarrow r_2$$
$$d_n \rightarrow r_n$$

where:

1. Each di is a new symbol, not in E and not the same as any other of the cTs, and

2. Each T{ is a regular expression over the alphabet E U {d\,d2,.. . ,

By restricting to E and the previously defined GTS, we avoid recursive definitions,and we can construct a regular expression over E alone, for each r$.

We do so by first replacing uses of d\ in r2 (which cannot use any of the d's except for d\), then replacing uses of d\ and d2 in r-$ by r\ and (the substituted) r2,and so on.

Finally, in rn we replace each di, for i — 1,2,... ,n — 1, by the substituted version of r$, each of which has only symbols of E.

Example 4 : C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

$$\textit{letter-} \rightarrow \mathbf{A} \mid \mathbf{B} \mid \cdots \mid \mathbf{Z} \mid \mathbf{a} \mid \mathbf{b} \mid \cdots \mid \mathbf{z} \mid \_$$
$$\textit{digit} \rightarrow \mathbf{0} \mid \mathbf{1} \mid \cdots \mid \mathbf{9}$$
$$\textit{id} \rightarrow \textit{letter-} (\textit{letter-} \mid \textit{digit})^*$$

Example 5 : Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4. The regular definition

$$\begin{aligned}
\textit{digit} &\rightarrow \mathbf{0} \mid \mathbf{1} \mid \cdots \mid \mathbf{9} \\
\textit{digits} &\rightarrow \textit{digit}\ \textit{digit}^* \\
\textit{optionalFraction} &\rightarrow . \ \textit{digits} \mid \mathbf{e} \\
\textit{optionalExponent} &\rightarrow (\mathbf{E} (+ \mid - \mid \mathbf{e}) \ \textit{digits}) \mid \mathbf{e} \\
\textit{number} &\rightarrow \textit{digits}\ \ \textit{optionalFraction}\ \ \textit{optionalExponent}
\end{aligned}$$

is a precise specification for this set of strings. That is, an optionalFraction is either a decimal point (dot) followed by one or more digits, or it is missing (the empty string). An optionalExponent, if not missing, is the letter E followed by an optional + or - sign, followed by one or more digits. Note that at least one digit must follow the dot, so number does not match 1., but does match 1.0.

Extensions of Regular Expressions:

- Since Kleene introduced regular expressions with the basic operators for union, concatenation, and Kleene closure in the 1950s, many extensions have been added to regular expressions to enhance their ability to specify string patterns.
- Here we mention a few notational extensions that were first incorporated into Unix utilities such as Lex that are particularly useful in the specification lexical analyzers.

The references to this chapter contain a discussion of some regular expression variants in use today.

1. One or more instances.

- The unary, postfix operator + represents the positive closure of a regular expression and its language. That is, if r is a regular expression, then (r) + denotes the language (L(r)) + .
- The operator + has the same precedence and associativity as the operator *.
- Two useful algebraic laws, r* = r+\e and r+ = rr* = r*r relate the Kleene closure and positive closure.

2. Zero or one instance.

- The unary postfix operator ? means "zero or one occurrence."
- That is, r? is equivalent to r|e, or put another way, L(r?) = L(r) U {e}.
- The ? operator has the same precedence and associativity as * and +.

3. **Character classes.**

- A regular expression aifal • • • \an, where the a^s are each symbols of the alphabet, can be replaced by the shorthand [aia,2 • • - an].
- More importantly, when 0 1 , 0 2 , . . . , a n f ° r m a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by o i - a n , that is, just the first and last separated by a hyphen.
- Thus, [abc] is shorthand for a|b|c, and [a-z] is shorthand for a | b | . - - | z .

Example 6 : Using these shorthands, we can rewrite the regular definition of Example as:

```
letter.   ->   [A-Za-z_]
  digit   ->   [0-9]
     id   ->   letter- ( letter ⁞ digit )*
```

The regular definition of Example 5 can also be simplified:

```
   digit   ->   [0-9]
  digits   —>•  digit
number     -»•  digits (.  digits)?  ( E  [+-]?  digits )?
```

Below table shows Lex regular expressions:

EXPRESSION	MATCHES	EXAMPLE
c	the one non-operator character c	a
V	character c literally	*
llgll	string s literally	"**"
	any character but newline	a. *b
	beginning of a line	~abc
$	end of a line	abc$
[*]	any one of the characters in string *s*	[abc]
	any one character not in string *s*	[~abc]
r*	zero or more strings matching *r*	a*
r+	one or more strings matching *r*	a+
r?	zero or one r	a?
r{m, n}	between *m* and *n* occurrences of r	a[1,5]
rir₂	an ?*i followed by an r_2	ab
r\ | r_2	an ri or an r_2	a|b
(r)	same as r	(a|b)
ri/r_2	7*1 when followed by r_2	abc/123

Following Figure shows: Filename expressions used by the shell command sh

EXPRESSION	MATCHES	EXAMPLE
v	string *s* literally	'V
V	character c literally	V
*	any string	*.o
?	any character	sort1.?
[s]	any character in *s*	sort1.[cso]

2.4 FINITE AUTOMATA

We shall now discover how Lex turns its input program into a lexical analyzer.

At the heart of the transition is the formalism known as finite automata.

These are essentially graphs, like transition diagrams, with a few differences:

1. Finite automata are recognizers; they simply say "yes" or "no" about each possible input string.

2. Finite automata come in two flavors:

   (a) Nondeterministic finite automata (NFA) have no restrictions on the labels of their edges.

   A symbol can label several edges out of the same state, and e, the empty string, is a possible label.

   (b) Deterministic finite automata (DFA) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

Both deterministic and nondeterministic finite automata are capable of recognizing the same languages.

In fact these languages are exactly the same languages, called the regular languages, that regular expressions can describe.

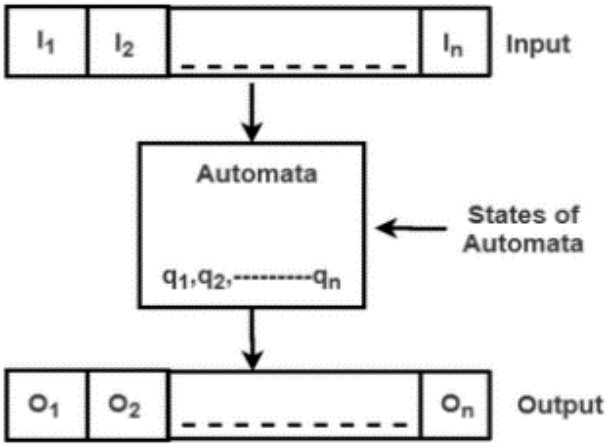


Figure: *Features of Finite Automata*

The above figure shows the following features of automata:

1. Input
2. Output
3. States of automata
4. State relation
5. Output relation

A Finite Automata consists of the following:

Q : Finite set of states.

? : set of Input Symbols.

q : Initial state.

F : set of Final States.

? : Transition Function.

Formal specification of machine is { Q, ?, q, F, ? }

FA is characterized into two types:

1. **Deterministic Finite Automata (DFA)**
2. **Nondeterministic Finite Automata(NFA)**

1) Deterministic Finite Automata (DFA):

DFA consists of 5 tuples {Q, ?, q, F, ?}.

Q : set of all states.

? : set of input symbols. (Symbols which machine takes as input)

q : Initial state. (Starting state of a machine)

F : set of final state.

? : Transition Function, defined as ? : Q X ? --> Q.

In a DFA, for a particular input character, the machine goes to one state only.

A transition function is defined on every state for every input symbol. Also in DFA null (or ?) move is not allowed, i.e., DFA cannot change state without any input character.

For example, construct a DFA which accept a language of all strings ending with ‘a’.

Given: ? = {a,b}, q = {q0}, F={q1}, Q = {q0, q1}

First, consider a language set of all the possible acceptable strings in order to construct an accurate state transition diagram.

L = {a, aa, aaa, aaaa, aaaaa, ba, bba, bbbaa, aba, abba, aaba, abaa}

Above is simple subset of the possible acceptable strings there can many other strings which ends with ‘a’ and contains symbols {a,b}.

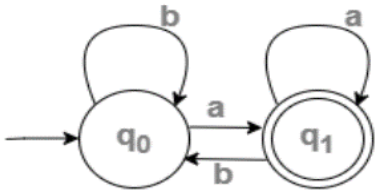


Fig 1. State Transition Diagram for DFA with ? = {a, b}

Strings not accepted are,

ab, bb, aab, abbb, etc.

State transition table for above automaton,

?State \ Symbol?	a	b
q_0	q_1	q_0
q_1	q_1	q_0

One important thing to note is, there can be many possible DFAs for a pattern.

A DFA with a minimum number of states is generally preferred.

2) Nondeterministic Finite Automata(NFA):

NFA is similar to DFA except following additional features:

- Null (or ?) move is allowed i.e., it can move forward without reading symbols.
- Ability to transmit to any number of states for a particular input.

However, these above features don't add any power to NFA.

If we compare both in terms of power, both are equivalent.

Due to the above additional features, NFA has a different transition function, the rest is the same as DFA.

?: Transition Function

?: Q X (? U ?) --> 2 ^ Q.

As you can see in the transition function is for any input including null (or ?), NFA can go to any state number of states.

For example, below is an NFA for the above problem.

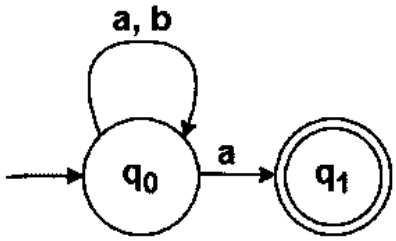


Fig 2. State Transition Diagram for NFA with ? = {a, b}

State Transition Table for above Automaton,

?State \ Symbol?	a	b
q_0	$\{q_0, q_1\}$	q_0
q_1	?	?

One important thing to note is, in NFA, if any path for an input string leads to a final state, then the input string is accepted.

For example, in the above NFA, there are multiple paths for the input string "00".

Since one of the paths leads to a final state, "00" is accepted by the above NFA.

Take Note:

Since all the tuples in DFA and NFA are the same except for one of the tuples, which is Transition Function (?)

In case of DFA

? : Q X ? --> Q

In case of NFA

? : Q X ? --> 2Q

Now if you observe you'll find out Q X ? –> Q is part of Q X ? –> 2Q.

On the RHS side, Q is the subset of 2Q which indicates Q is contained in 2Q or Q is a part of 2Q, However, the reverse isn't true.

So mathematically, we can conclude that **every DFA is NFA but not vice-versa**.

Yet there is a way to convert an NFA to DFA, so there exists an equivalent DFA for every NFA.

Important Points to Remember:

1. Both NFA and DFA have the same power and each NFA can be translated into a DFA.
2. There can be multiple final states in both DFA and NFA.
3. NFA is more of a theoretical concept.
4. DFA is used in Lexical Analysis in Compiler.
5. If the number of states in the NFA is N then, its DFA can have maximum 2N number of states.

NONDETERMINISTIC FINITE AUTOMATA (NFA)

A nondeterministic finite automaton (NFA) consists of:

1. A finite set of states 5.
2. A set of input symbols E, the input alphabet. We assume that e, which stands for the empty string, is never a member of E.
3. A transition function that gives, for each state, and for each symbol in E U {e} a set of next states.
4. A state so from S that is distinguished as the start state (or initial state).
5. A set of states F, a subset of S, that is distinguished as the accepting states (or final states).

We can represent either an NFA or DFA by a transition graph, where the nodes are states and the labeled edges represent the transition function.

There is an edge labeled from state s to state t if and only if t is one of the next states for state s and input a. This graph is very much like a transition diagram, except:

a) The same symbol can label edges from one state to several different states, and

b) An edge may be labeled by e, the empty string, instead of, or in addition to, symbols from the input alphabet.

Example : The transition graph for an NFA recognizing the language of regular expression (a|b)*abb is shown below Fig.

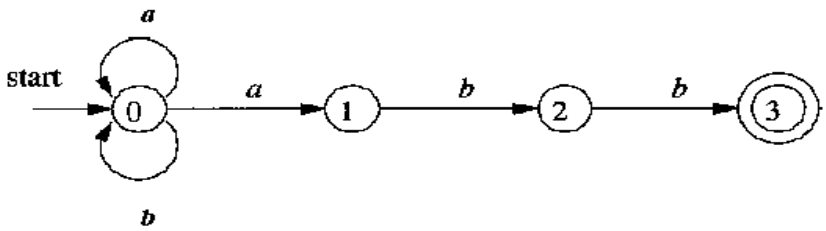


Figure 3.24: A nondeterministic finite automaton

This abstract example, describing all strings of a's and &'s ending in the particular string abb, will be used throughout this section.

It is similar to regular expressions that describe languages of real interest, however.

For instance, an expression describing all files whose name ends in .o is any*.o, where any stands for any printable character.

As per transition diagrams, the double circle around state 3 indicates that this state is accepting.

Notice that the only ways to get from the start state 0 to the accepting state is to follow some path that stays in state 0 for a while, then goes to states 1, 2, and 3 by reading abb from the input.

Thus, the only strings getting to the accepting state are those that end in abb.

Transition Tables

We can also represent an NFA by a transition table, whose rows correspond to states, and whose columns correspond to the input symbols and e.

The entry for a given state and input is the value of the transition function applied to those arguments. If the transition function has no information about that state-input pair, we put 0 in the table for the pair.

Example : The transition table for the NFA of above Fig. 3.24 is shown below Fig. 3.25.

STATE	*a*	*b*	*e*
0	{0,1}	{0}	0
1	0	{2}	0
2	0	{3}	0
3	0	0	0

Figure 3.25: Transition table for the NFA of Fig. 3.24

The transition table has the advantage that we can easily find the transitions on a given state and input. Its disadvantage is that it takes a lot of space, when the input alphabet is large, yet most states do not have any moves on most of the input symbols.

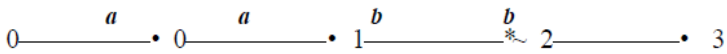
Acceptance of Input Strings by Automata

An NFA accepts input string x if and only if there is some path in the transition graph from the start state to one of the accepting states, such that the symbols along the path spell out x.

Note that e labels along the path are effectively ignored, since the empty string does not contribute to the string constructed along the path.

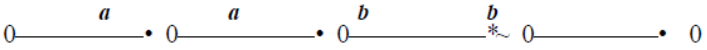
Example: The string aabb is accepted by the NFA of Fig. 3 . 2 4 .

The path labeled by aabb from state 0 to state 3 demonstrating this fact is:



Note that several paths labeled by the same string may lead to different states.

For instance, path is another path from state 0 labeled by the string aabb.



This path leads to state 0, which is not accepting.

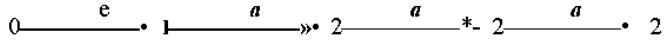
However, remember that an NFA accepts a string as long as some path labeled by that string leads from the start state to an accepting state.

The existence of other paths leading to a nonaccepting state is irrelevant.

The language defined (or accepted) by an NFA is the set of strings labeling some path from the start to an accepting state.

As was mentioned, the NFA of Fig. 3 . 2 4 defines the same language as does the regular expression (a|b)*abb,that is, all strings from the alphabet {a, b} that end in abb. We may use L(A) to stand for the language accepted by automaton A.

Example 3.17: Figure 3.26 is an NFA accepting L(**aa*|bb***). String *aaa* is accepted because of the path



Note that e's "disappear" in a concatenation, so the label of the path is *aaa*.

Deterministic Finite Automata:

A deterministic finite automaton (DFA) is a special case of an NFA where:

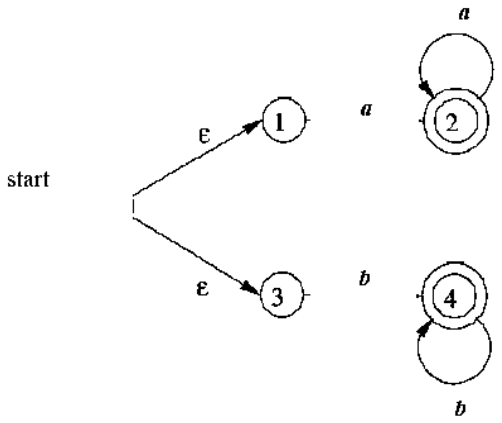


Figure 3.26: NFA accepting **aa*|bb***

1. There are no moves on input e, and
2. For each state s and input symbol a, there is exactly one edge out of s

If we are using a transition table to represent a DFA, then each entry is a single state, we may therefore represent this state without the curly braces that we use to form sets.

While the NFA is an abstract representation of an algorithm to recognize the strings of a certain language, the DFA is a simple, concrete algorithm for recognizing strings. It is fortunate indeed that every regular expression and every NFA can be converted to a DFA accepting the same language, because it is the DFA that we really implement or simulate when building lexical analyzers.

The following algorithm shows how to apply a DFA to a string.

Algorithm: Simulating a DFA.

INPUT : An input string x terminated by an end-of-file character eof.

A DFA D with start state so, accepting states F, and transition function move.

OUTPUT : Answer "yes" if D accepts x; "no" otherwise.

METHOD : Apply the algorithm in Fig. 3.27 to the input string x.

The function move(s,c) gives the state to which there is an edge from state s on input c.

The function next Char returns the next character of the input string x.

```
1)  S = e-closure(so);
2)  c = nextCharQ;
3)  while ( c != eof) {
4)          S = e-closure(move(S,c))
5)          c = nextCharQ;
6)  }
7)  if ( 5 n F != 0 ) return "yes";
8)  else return "no";
```

Figure 3.37: Simulating an NFA

Algorithm : The subset construction of a DFA from an NFA.

INPUT : An NFA JV.

OUTPUT A DFA D accepting the same language as N.

METHOD : Our algorithm constructs a transition table Dtran for D.

Each state of D is a set of NFA states, and we construct Dtran so D will simulate

"**in parallel**" all possible moves N can make on a given input string.

Our first problem is to deal with e-transitions of N properly.

Note that s is a single state of N, while T is a set of states of N.

Example 3.19: In Fig. 3.28 we see the transition graph of a DFA accepting the language (a|b)*abb, the same as that accepted by the NFA of Fig. 3.24.

Given the input string ababb, this DFA enters the sequence of states 0 , 1 , 2 , 1 , 2 ,3 and returns "yes."

```
s = s0
c = nextCharQ;
while ( c != eof) {
        s = move(s,c);
        c = nextCharQ;
}
if ( s is in F ) return "yes";
else return "no";
```

Figure 3.27: Simulating a DFA

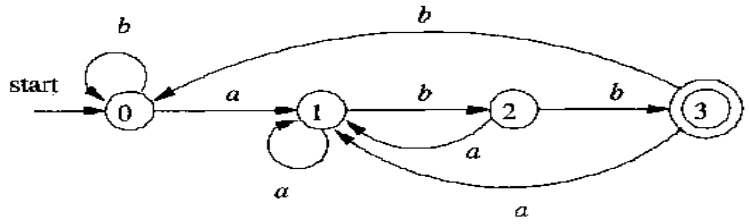


Figure 3.28: DFA accepting (a|b)*abb

Regular Expressions to Automata

- The regular expression is the notation of choice for describing lexical analyzers and other pattern-processing software.
- However, implementation of that software requires the simulation of a DFA, as in Algorithm, or perhaps simulation of an NFA.
- Because an NFA often has a choice of move on an input symbol (as Fig. 3.24 does oh input a from state 0) or on e (as Fig. 3.26 does from state 0), or even a choice of making a transition on e or on a real input symbol, its simulation is less straightforward than for a DFA.
- Thus often it is important to convert an NFA to a DFA that accepts the same language.
- In this section we shall first show how to convert NFA's to DFA's.
- Then, we use this technique, known as "**the subset construction**," to give a useful algorithm for simulating NFA's directly, in situations (other than lexical analysis) where the NFA-to-DFA conversion takes more time than the direct simulation.
- Next, we show how to convert regular expressions to NFA's, from which a DFA can be constructed if desired.
- We conclude with a discussion of the time-space tradeoffs inherent in the various methods for implementing regular expressions, and see how to choose the appropriate method for your application.

Operations on NFA States:

OPERATION	DESCRIPTION
e-closure(s)	Set of NFA states reachable from NFA state *s* on e-transitions alone.
e-closure(T)	Set of NFA states reachable from some NFA state *s* in set *T* on e-transitions alone; = U, i, T *e-closure(s).*
move(T, a)	Set of NFA states to which there is a transition on input symbol *a* from some state *s* in *T.*

Conversion of an NFA to a DFA

The general idea behind the subset construction is that each state of the constructed DFA corresponds to a set of NFA states.

An NFA can have zero, one or more than one move from a given state on a given input symbol.

An NFA can also have NULL moves (moves without input symbol).

On the other hand, DFA has one and only one move from a given state on a given input symbol.

Steps for converting NFA to DFA:

Step 1: Convert the given NFA to its equivalent transition table

To convert the NFA to its equivalent transition table, we need to list all the states, input symbols, and the transition rules.

The transition rules are represented in the form of a matrix, where the rows represent the current state, the columns represent the input symbol, and the cells represent the next state.

Step 2: Create the DFA's start state

The DFA's start state is the set of all possible starting states in the NFA.

This set is called the "epsilon closure" of the NFA's start state.

The epsilon closure is the set of all states that can be reached from the start state by following epsilon (?) transitions.

Step 3: Create the DFA's transition table

The DFA's transition table is similar to the NFA's transition table, but instead of individual states, the rows and columns represent sets of states.

For each input symbol, the corresponding cell in the transition table contains the epsilon closure of the set of states obtained by following the transition rules in the NFA's transition table.

Step 4: Create the DFA's final states

The DFA's final states are the sets of states that contain at least one final state from the NFA.

Step 5: Simplify the DFA

The DFA obtained in the previous steps may contain unnecessary states and transitions.

To simplify the DFA, we can use the following techniques:

- Remove unreachable states: States that cannot be reached from the start state can be removed from the DFA.
- Remove dead states: States that cannot lead to a final state can be removed from the DFA.
- Merge equivalent states: States that have the same transition rules for all input symbols can be merged into a single state.

Step 6: Repeat steps 3-5 until no further simplification is possible

After simplifying the DFA, we repeat steps 3-5 until no further simplification is possible.

The final DFA obtained is the minimized DFA equivalent to the given NFA.

Example: Consider the following NFA shown in Figure 1.

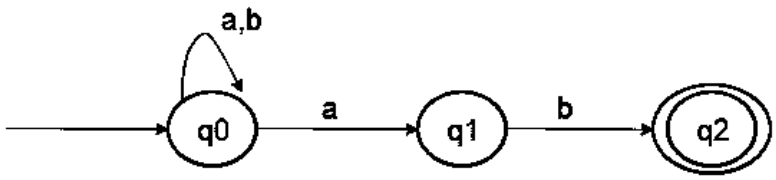


Figure 1

Following are the various parameters for NFA. Q = { q0, q1, q2 } ? = (a, b)
F = { q2 } ? (Transition Function of NFA)

State	a	b
q0	**q0,q1**	**q0**
q1		**q2**
q2		

Step 1:
Q' = ?

Step 2:
Q' = {q0}

Step 3:
For each state in Q', find the states for each input symbol.

Currently, state in Q' is q0, find moves from q0 on input symbol a and b using transition function of NFA and update the transition table of DFA. ?' (Transition Function of DFA)

State	a	b
q0	**{q0,q1}**	**q0**

Now { q0, q1 } will be considered as a single state.

As its entry is not in Q', add it to Q'.

So Q' = { q0, { q0, q1 } } Now, moves from state { q0, q1 } on different input symbols are not present in transition table of DFA, we will calculate it like: ?' ({ q0, q1 }, a) = ? (q0, a) ? ? (q1, a) = { q0, q1 } ?' ({ q0, q1 }, b) = ? (q0, b) ? ? (q1, b) = { q0, q2 }

Now we will update the transition table of DFA. ?' (Transition Function of DFA)

State	a	B
q0	**{q0,q1}**	**q0**
{q0,q1}	**{q0,q1}**	**{q0,q2}**

Now { q0, q2 } will be considered as a single state.

As its entry is not in Q', add it to Q'. So Q' = { q0, { q0, q1 }, { q0, q2 } }

Now, moves from state {q0, q2} on different input symbols are not present in transition table of DFA, we will calculate it like: ?' ({ q0, q2 }, a) = ? (q0, a) ? ? (q2, a) = { q0, q1 } ?' ({ q0, q2 }, b) = ? (q0, b) ? ? (q2, b) = { q0 }

Now we will update the transition table of DFA. ?' (Transition Function of DFA)

State	a	B
q0	**{q0,q1}**	**q0**
{q0,q1}	**{q0,q1}**	**{q0,q2}**
{q0,q2}	**{q0,q1}**	**q0**

As there is no new state generated, we are done with the conversion.

Final state of DFA will be state which has q2 as its component i.e., { q0, q2 }

Following are the various parameters for DFA. Q' = { q0, { q0, q1 }, { q0, q2 } } ? = (a, b) F = { { q0, q2 } } and transition function ?' as shown above.

The final DFA for above NFA has been shown in Figure 2.

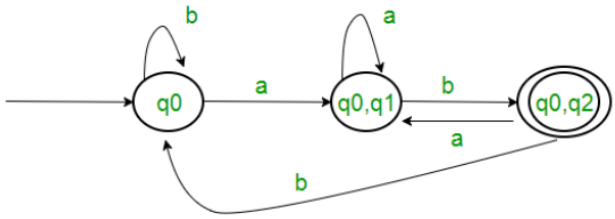


Figure 2

Note : Sometimes, it is not easy to convert regular expression to DFA. First you can convert regular expression to NFA and then NFA to DFA.

Example : The number of states in the minimal deterministic finite automaton corresponding to the regular expression (0 + 1)* (10) is _____.

Solution :

First, we will make an NFA for the above expression. To make an NFA for (0 + 1)*, NFA will be in same state q0 on input symbol 0 or 1.

Then for concatenation, we will add two moves (q0 to q1 for 1 and q1 to q2 for 0) as shown in Figure 3.

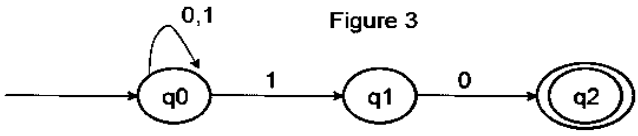


Figure 3

Using above algorithm, we can convert NFA to DFA as shown in Figure 4

State	0	1
q0	q0	q0,q1
q0,q1	q0,q2	q0,q1
q0,q2	q0	q0,q1

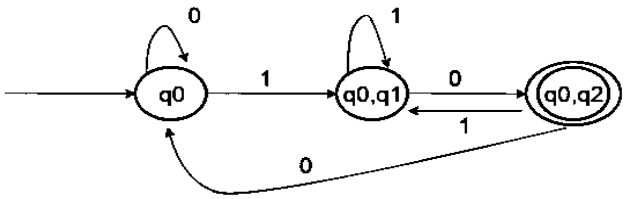


Figure 4

2.4 LEXICAL ANALYZER GENERATORS LEX:

In this section, we introduce a tool called Lex, or in a more recent implementation Flex, that allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens.

The input notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex compiler.

Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called l e x . y y . c, that simulates this transition diagram.

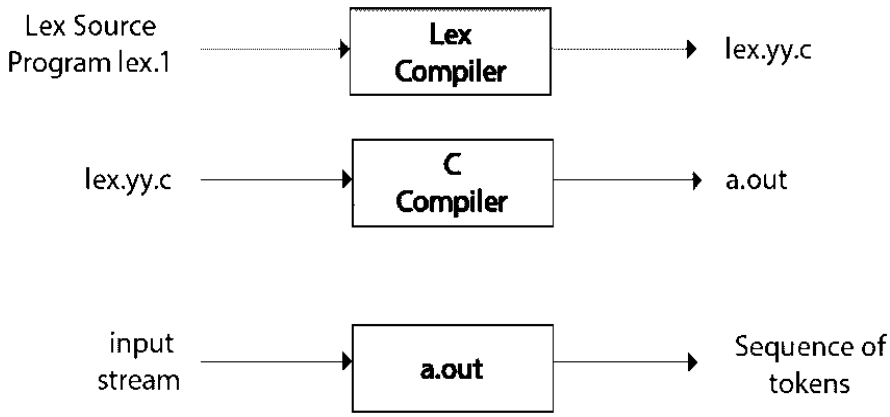
The mechanics of how this translation from regular expressions to transition diagrams occurs is the subject of the next sections; here we only learn the Lex language.

LEX

- Lex is a program that generates lexical analyzer.
- It is used with YACC parser generator.
- The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

The function of Lex is as follows:

- Firstly lexical analyzer creates a program lex.1 in the Lex language.
- Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
- Finally C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



USE OF LEX:

- Below Figure suggests how Lex is used.

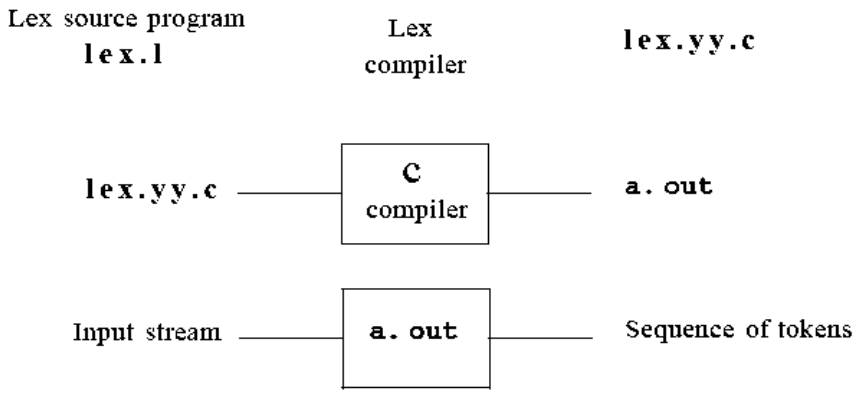


Figure: Creating a lexical analyzer with Lex

- An input file, which we call l e x . l , is written in the Lex language and describes the lexical analyzer to be generated.
- The Lex compiler transforms l e x . 1 to a C program, in a file that is always named l e x . y y . c.
- The latter file is compiled by the C compiler into a file called a . o u t , as always.
- The C-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens.
- The normal use of the compiled C program, referred to as a. out in above Fig.,is as a subroutine of the parser.
- It is a C function that returns an integer, which is a code for one of the possible token names.
- The attribute value, whether it be another numeric code, a pointer to the symbol table, or nothing, is placed in a global variable y y l v a l , 2 which is shared between the lexical analyzer and parser, thereby making it simple to return both the name and an attribute value of a token.

Structure of Lex Programs:

- A Lex program has the following form:

```
declarations
%%
translation rules
%%
auxiliary functions
```

- The declarations section includes declarations of variables, manifest constants (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions.

- The translation rules each have the form
  Pattern { Action }
- Each pattern is a regular expression, which may use the regular definitions of the declaration section.
- The actions are fragments of code, typically written in C, although many variants of Lex using other languages have been created.
- The third section holds whatever additional functions are used in the actions.
- Alternatively, these functions can be compiled separately and loaded with the lexical analyzer.
- The lexical analyzer created by Lex behaves in concert with the parser as follows.
- When called by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it finds the longest prefix of the input that matches one of the patterns Pi.
- It then executes the associated action Ai.
- Typically, Ai will return to the parser, but if it does not (e.g., because Pi describes whitespace or comments), then the lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser.
- The lexical analyzer returns a single value, the token name, to the parser, but uses the shared, integer variable y y l v a l to pass additional information about the lexeme found, if needed.

Lex File Format

- A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:
  { definitions }
  %%
  { rules }
  %%
  { user subroutines }
- Definitions include declarations of constant, variable and regular definitions.
- Rules define the statement of form p1 {action1} p2 {action2}....pn {action}.
- Where pi describes the regular expression and action1 describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.

Lex And Yacc.

- If you want to use Lex with Yacc, note that what Lex writes is a program named yylex(), the name required by Yacc for its analyzer.
- Normally, the default main program on the Lex library calls this routine, but if Yacc is loaded, and its main program is used, Yacc will call yylex().
- In this case each Lex rule should end with return(token);
  where the appropriate token value is returned.
- An easy way to get access to Yacc's names for tokens is to compile the Lex output file as part
- of the Yacc output file by placing the line # include "lex.yy.c" in the last section of Yacc input.
- Supposing the grammar to be named ``good" and the lexical rules to be named ``better" the UNIX command sequence can just be:

```
yacc good
lex better
cc y.tab.c -ly -ll
```

- The Yacc library (-ly) should be loaded before the Lex library, to obtain a main program which invokes the Yacc parser.
- The generations of Lex and Yacc programs can be done in either order.

Points to Remember:

- The general form of a Lex source file is:
  {definitions}
  %%
  {rules}
  %%
  {user subroutines}
- The definitions section contains a combination of:
- Definitions, in the form ``name space translation".
- Included code, in the form ``space code".
- Included code, in the form
  %{
  code
  %}
- Start conditions, given in the form
  %S name1 name2 ...

- Character set tables, in the form
  %T
  number space character-string
  ...
  %T

- Changes to internal array sizes, in the form

  %x nnn

  where nnn is a decimal integer representing an array size and x selects the parameter as follows:

```
Letter          Parameter
  p       positions
  n       states
  e       tree nodes
  a       transitions
  k       packed character classes
  o       output array size
```

- Lines in the rules section have the form ``expression action" where the action may be continued on succeeding lines by using braces to delimit it.

2.5 SUMMARY

In this chapter we have learn about fundamental of lexical analysis. What is Lexical analysis And working of it.We discussed Finite automata ,learn the types if FA. and conversion of DFA and NFA.

Brief Introduction to regular expression is given.

2.6 EXERCISE

Answer the following:

1. Explain features of DFA and NFA.
2. Identify the interactions between the lexical analyzer and the parser.
3. Explain regular expressions with examples.
4. Explain the role of Lexical analysis
5. Write the steps to convert Non-Deterministic Finite Automata (NDFA) into Deterministic Finite Automata (DFA).
6. Construct its equivalent DFA.

   Let M=({q0,q1}, {0,1}, δ, q0, {q1}).
   Be NFA where δ(q0,0)={q0,q1}, δ(q1,1) = {q1}
   δ(q1, 0)=φ, δ(q1, 1)={q0, q1}

7. Convert the given NFA to DFA:

Input/State	0	1
➔ q0	{q0, q1}	q0
q1	q2	q1
q2	q3	q3
q3 (final state)	φ (null character)	q2

8. What is Regular Expression? Write the regular expression for:

   a. R=R1+R2 (Union operation)

   b. R=R1.R2 (concatenation Operation)

   c. R=R1* (Kleen Closure)

   d. R=R+ (Positive Clouser)

   e. Write a regular expression for a language containing strings which end with "abb" over Σ= {a,b}.

   f. Construct a regular expression for the language containing all strings having any number of a's and b's except the null string.

9. Construct Deterministic Finite Automata to accept the regular expression :

   (0+1)* (00+11) (0+1)*

10. Define regular expression and draw the transition diagram for the following expression:

    a. ab*cbb

    b. $(0^* + 1) . (01^*)$

11. Develop the Structure of lex program.

12. What is NFA? And discuss with examples (a/b)*

13. Define lex and give its execution steps.

14. Outline the role of lexical analysis in compiler design.

15. Discuss in detail about the role of Lexical analyzer with the possible error recovery schemes.

16. Describe in detail about issues in lexical analysis.

17. Define Finite Automata. Differentiate Deterministic Finite Automata and Non-Deterministic Finite Automata with examples.

18. Solve the given regular expression into NFA using Thompson construction

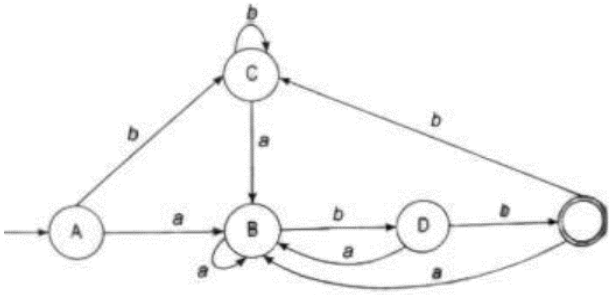
    i) (a/b)* abb (a/b)*.

    ii) ab*/ab

19. Create DFA the following regular expression.(a/b)*abb.

20. Illustrate the algorithm for minimizing the number of states of a DFA.

21. Minimize the following states of DFA



22. Define Lex and Lex specifications. How lexical analyzer is constructed using lex? Give an example.
23. Explain the lex program for tokens. Describe in detail the tool for generating lexical analyzer.
24. Find the NFA for the given regular expression and find the minimized DFA for the constructed NFA..(a/b)*(a/b)
25. (i) Create languages denoted by the following regular expressions

    a) (a|b)*a(a|b)(a|b)

    b) a*ba*ba*ba*

    (ii) Write regular definitions for the following languages:
    a) All strings of lowercase letters that contain the five vowels in order.
    b) All strings of lowercase letters in which the letters are in ascending lexicographic order.
26. Find transition diagrams for the following regular expression and regular definition.

    a(a|b)*a ((ε|a)b*)*

    a. All strings of digits with at most one repeated digit.

    b. All strings of a's and b's that do not contain the substring abb.

    c. All strings of a's and b's that do not contain the subsequence abb.
27. Explain in detail the tool for generating Lexical-Analyzer with an example program.
28. Develop the Lex Program to recognize the identifiers, constants and operators.

2.7 REFERENCES

Compilers: Principles, Techniques, and Tools" by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman 2nd Edition, Pearson Publication, 2006 ISBN-13: 978-0321486813

https://www.javatpoint.com/lex

https://www.geeksforgeeks.org/conversion-from-nfa-to-dfa/

❄❄❄❄❄❄❄

3

SYNTAX ANALYSIS

Unit Structure

3.0 Objective

3.1 Introduction

3.2 The Role of the Parser

- 3.2.1 Syntax Error Handling
- 3.2.2 Error-Recovery Strategies

3.3 Context-free Grammars

- 3.3.1 The Formal Definition of a Context-Free Grammar
- 3.3.2 Notational Conventions
- 3.3.3 Derivations

3.4 Top-Down Parsing (LL Parsing)

- 3.4.1 recursive-descent parsing
- 3.4.2 First and Follow

3.5 Bottom-Up Parsing

- 3.5.1 Reductions
- 3.5.2 Handle Pruning

3.6 Syntax analyzer generators

- 3.6.1 Parser Generator YACC
- 3.6.2 The Translation Rules Part
- 3.6.3 Using Yacc with Ambiguous Grammars
- 3.6.4 Error Recovery in YACC

3.7 Summary

3.8 Review Questions

3.0 OBJECTIVES

1. Learn the function of parsers and how to handle and recover from syntax errors.
2. Understand the formal definition and usage of context-free grammars in representing programming language syntax.
3. Master top-down (LL) and bottom-up parsing methods, including recursive-descent parsing and reduction processes.

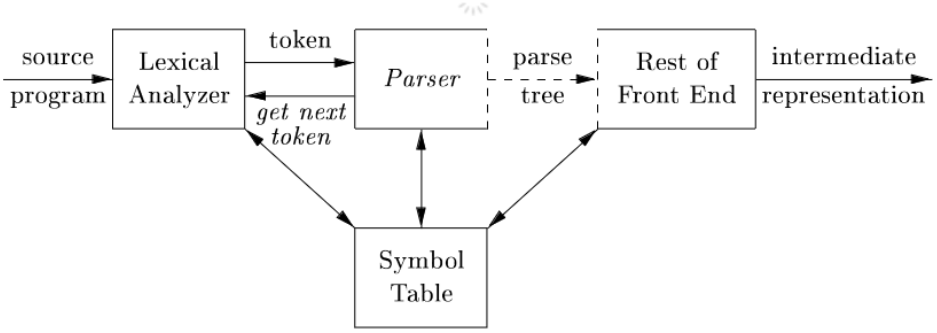
4. Gain proficiency in using parser generators like YACC, including managing translation rules and error recovery.

3.1 INTRODUCTION

We explore how the parser integrates into a standard compiler. Following that, we examine common grammars used for arithmetic expressions. These grammars are sufficient to demonstrate the core principles of parsing because the techniques applicable to expressions extend to most programming constructs. The section concludes with a discussion on error handling, highlighting the parser's need to respond appropriately when it encounters input that cannot be produced by its grammar.

3.2 THE ROLE OF THE PARSER

In our compiler model, the parser receives a sequence of tokens from the lexical analyzer. It verifies that this sequence can be produced by the grammar of the source language. The parser is expected to report syntax errors clearly and recover from common errors to continue processing the rest of the program. For well-formed programs, the parser conceptually builds a parse tree and sends it to the rest of the compiler for further processing. However, the parse tree doesn't need to be explicitly constructed, as checking and translation actions can occur during parsing. Consequently, the parser and the rest of the front end could be implemented as a single module.



There are three main types of parsers for grammars: universal, top-down, and bottom-up. Universal parsing techniques, such as the Cocke-Younger-Kasami algorithm and Earley's algorithm, can handle any grammar but are too inefficient for use in production compilers.

Commonly used parsing methods in compilers fall into two categories: top-down and bottom-up. Top-down parsers build the parse tree from the root down to the leaves, while bottom-up parsers start from the leaves and build up to the root. In both methods, the input is processed from left to right, one symbol at a time.

The most efficient top-down and bottom-up parsing methods are limited to specific subclasses of grammars, but LL and LR grammars, in particular, are powerful enough to describe most syntactic constructs found in modern

programming languages. LL grammars are often used in hand-crafted parsers, such as those using predictive parsing techniques. LR grammar parsers are typically generated using automated tools

We assume that the parser produces a representation of the parse tree for the token stream received from the lexical analyzer. In practice, several tasks might be performed during parsing, such as collecting information about various tokens into the symbol table, performing type checking and other semantic analyses, and generating intermediate code.

3.2.1 Syntax Error Handling

Two specific strategies, panic-mode and phrase-level recovery, are discussed in more detail in relation to specific parsing methods.

If compilers only had to process correct programs, their design and implementation would be greatly simplified. However, compilers are expected to help programmers locate and fix errors that inevitably occur despite their best efforts. Interestingly, few programming languages are designed with error handling in mind, even though errors are common. Our world would be vastly different if spoken languages required the same level of syntactic accuracy as programming languages. Most programming language specifications do not describe how a compiler should respond to errors; this is left to the compiler designer. Planning error handling from the start can simplify the compiler's structure and improve its error-handling capabilities.

Common programming errors can occur at various levels:

- Lexical errors include misspellings of identifiers, keywords, or operators (e.g., using "elipseSize" instead of "ellipseSize") and missing quotes around strings.
- Syntactic errors include misplaced semicolons or extra/missing braces. For example, a "case" statement without an enclosing "switch" in C or Java is a syntactic error, though this is often caught later in the compilation process.
- Semantic errors include type mismatches between operators and operands, such as returning a value in a Java method with a void return type.
- Logical errors involve incorrect reasoning by the programmer or misuse of operators, such as using "=" instead of "==" in C. Although syntactically correct, this might not reflect the programmer's intent.

Parsing methods are precise enough to detect syntactic errors efficiently. Methods like LL and LR detect errors as soon as the token stream cannot be parsed further according to the grammar. They have the "viable-prefix" property, meaning they detect errors as soon as an incomplete prefix is encountered.

Emphasizing error recovery during parsing is crucial because many errors

appear syntactic and are exposed when parsing cannot continue. While some semantic errors, like type mismatches, can be detected efficiently, accurately identifying semantic and logical errors at compile time is generally difficult.

The error handler in a parser has several key goals:

- Report errors clearly and accurately.
- Recover from errors quickly to detect subsequent errors.
- Minimize overhead when processing correct programs.

Common errors are usually simple, so a straightforward error-handling mechanism often suffices.

To report errors effectively, the error handler must indicate where the error was detected in the source program, as the actual error likely occurred within the previous few tokens. A common strategy is to print the offending line and point to the error's location.

3.2.2 Error-Recovery Strategies

Once an error is detected, how should the parser recover? Although no single strategy is universally effective, several methods have broad applicability. The simplest approach is for the parser to halt with an informative error message upon detecting the first error. However, more errors can be identified if the parser can recover to a state where it can continue processing the input with the hope of providing meaningful diagnostic information. If errors accumulate excessively, the compiler should stop after reaching a certain error limit to avoid overwhelming the user with numerous "spurious" errors.

The following recovery strategies are discussed in detail: panic-mode, phrase-level, error productions, and global correction.

Panic-Mode Recovery

In this method, upon encountering an error, the parser discards input symbols one at a time until it finds one of a set of designated synchronizing tokens. These tokens are usually delimiters, such as semicolons or closing braces, which have clear and unambiguous roles in the source program. The choice of synchronizing tokens depends on the source language. While panic-mode recovery may skip a substantial portion of the input without checking for additional errors, it is simple and, unlike some other methods, guarantees not to enter an infinite loop.

Phrase-Level Recovery

When an error is found, the parser performs a local correction on the remaining input. This involves replacing a prefix of the remaining input with a string that allows the parser to continue. Typical local corrections include replacing a comma with a semicolon, deleting an extraneous semicolon, or inserting a missing semicolon. The choice of correction is left

to the compiler designer. It is crucial to choose replacements that do not lead to infinite loops, such as always inserting something before the current input symbol. Phrase-level replacement has been used in several error-repairing compilers as it can correct any input string. However, its major drawback is its difficulty in dealing with errors that occurred before the point of detection.

Error Productions

By anticipating common errors, the grammar for the language can be augmented with productions that generate erroneous constructs. A parser built from such an augmented grammar detects anticipated errors when an error production is used during parsing. The parser can then generate appropriate error diagnostics about the recognized erroneous construct.

Global Correction

Ideally, a compiler should make as few changes as possible when processing an incorrect input string. Algorithms exist to choose a minimal sequence of changes to obtain a globally least-cost correction. Given an incorrect input string x and a grammar G, these algorithms find a parse tree for a related string y, minimizing the number of insertions, deletions, and changes of tokens required to transform x into y. Unfortunately, these methods are generally too costly to implement in terms of time and space, so they remain mostly of theoretical interest.

It's important to note that the closest correct program might not align with the programmer's intent. Nevertheless, the concept of least-cost correction provides a standard for evaluating error-recovery techniques and has been used to find optimal replacement strings for phrase-level recovery.

3.3 Context-free Grammars

Grammars are used to systematically describe the syntax of programming language constructs like expressions and statements. For instance, using a syntactic variable stmt to denote statements and expr to denote expressions, the production:

stmt -> if (expr) stmt else stmt

specifies the structure of a conditional statement. Other productions then define precisely what an expr is and what else a stmt can be.

This section reviews the definition of a context-free grammar and introduces terminology for discussing parsing. The concept of derivations is particularly useful for understanding the order in which productions are applied during parsing

3.3.1 The Formal Definition of a Context-Free Grammar

A context-free grammar (or grammar for short) consists of terminals, nonterminal, a start symbol, and productions.

1. Terminals are the basic symbols from which strings are formed. They

are the actual tokens output by the lexical analyzer. For example, in the context of the if-else statement, the terminals might be the keywords if and else, and the symbols (and).

2. Nonterminal are syntactic variables that represent sets of strings. They help define the language generated by the grammar and impose a hierarchical structure on it. In the if-else statement, stmt and expr are nonterminals.

3. The start symbol is a distinguished nonterminal whose set of strings defines the language generated by the grammar. Conventionally, the productions for the start symbol are listed first.

4. Productions specify how terminals and nonterminals can be combined to form strings. Each production consists of:

   - A nonterminal called the head or left side, which defines some of the strings denoted by the nonterminal.
   - An arrow (-> or ::=) to separate the head from the body.
   - The body or right side, which consists of zero or more terminals and nonterminals. The components of the body describe one way in which strings of the nonterminal at the head can be constructed.

For example, in a grammar for simple arithmetic expressions, the terminals might be id, +, -, *, /, (, and). The nonterminals might be expression, term, and factor, with expression as the start symbol.

$$
\begin{aligned}
\mathit{expression} &\rightarrow \mathit{expression} + \mathit{term} \\
\mathit{expression} &\rightarrow \mathit{expression} - \mathit{term} \\
\mathit{expression} &\rightarrow \mathit{term} \\
\mathit{term} &\rightarrow \mathit{term} * \mathit{factor} \\
\mathit{term} &\rightarrow \mathit{term} / \mathit{factor} \\
\mathit{term} &\rightarrow \mathit{factor} \\
\mathit{factor} &\rightarrow (\mathit{expression}) \\
\mathit{factor} &\rightarrow \mathbf{id}
\end{aligned}
$$

Figure: Grammar for simple arithmetic expressions

3.3.2 Notational Conventions

To avoid repetitive statements about terminals, nonterminals, etc., the following notational conventions for grammars will be used throughout the remainder of this book:

1. Terminals:

   - Lowercase letters early in the alphabet (a, b, c).
   - Operator symbols such as +, *, etc.
   - Punctuation symbols such as parentheses, comma, etc.

- Digits 0 to 9.
- Boldface strings like id or if, each representing a single terminal symbol.

2. Nonterminals:
   - Uppercase letters early in the alphabet (A, B, C).
   - The letter S, usually representing the start symbol.
   - Lowercase, italic names like expr or stmt.
   - Uppercase letters late in the alphabet (X, Y, Z) represent grammar symbols (either nonterminals or terminals).
3. Lowercase letters late in the alphabet represent (possibly empty) strings of terminals.
4. Lowercase Greek letters (α, β, etc.) represent (possibly empty) strings of grammar symbols. Thus, a generic production can be written as A -> α, where A is the head and α is the body.
5. Unless stated otherwise, the head of the production is the start symbol.

   Example: With these conventions, the grammar of Example below can be rewritten concisely

$$\begin{array}{lcl} E & \rightarrow & E + T \mid E - T \mid T \\ T & \rightarrow & T * F \mid T / F \mid F \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

The notational conventions indicate that E, T, and F are nonterminals, with E as the start symbol. All other symbols are terminals.

3.3.3 Derivations

The construction of a parse tree can be made precise by adopting a derivational view, where productions are treated as rewriting rules. Starting from the start symbol, each step of rewriting replaces a nonterminal with the body of one of its productions. This view corresponds to the top-down construction of a parse tree. The precision provided by derivations is particularly helpful when discussing bottom-up parsing. Bottom-up parsing is related to a class of derivations known as "rightmost" derivations, where the rightmost nonterminal is rewritten at each step.

For example, consider the following grammar with a single nonterminal E, which adds a production E -> E to the grammar.

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id}$$

3.4 TOP-DOWN PARSING (LL PARSING)

Top-down parsing can be seen as the task of building a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder. Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

For example, the sequence of parse trees in the figure, for the input id + id * id, represents a top-down parse according to the grammar provided.

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

This sequence of trees corresponds to a leftmost derivation of the input. At each step of a top-down parse, the main challenge is to determine the production to be applied for a nonterminal, say A. Once an A-production is chosen, the remainder of the parsing process involves matching the terminal symbols in the production body with the input string.

The section begins with a general form of top-down parsing called recursive-descent parsing, which may involve backtracking to find the correct A-production. A special case of recursive-descent parsing is predictive parsing, where no backtracking is required. Predictive parsing predicts the correct A-production by looking ahead at the input a fixed number of symbols, typically just one (i.e., the next input symbol).

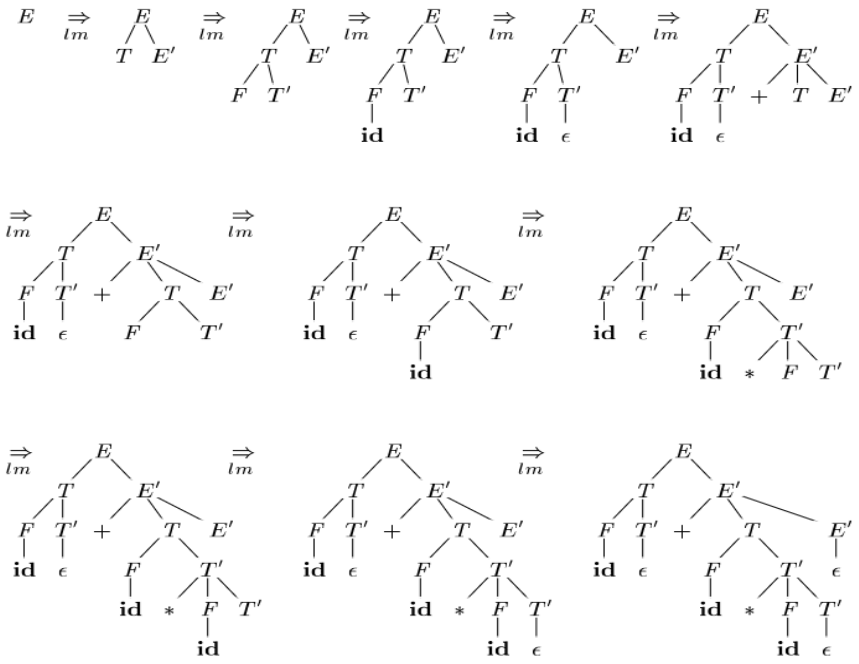


Figure: Top-down parse for id + id * id

For example, consider the top-down parse in the above figure, which constructs a tree with two nodes labeled E.

The class of grammars for which we can construct predictive parsers looking k symbols ahead in the input is sometimes called the LL(k) class. We will focus on the LL(1) class and introduce certain computations called FIRST and FOLLOW sets. From the FIRST and FOLLOW sets for a grammar, we can construct "predictive parsing tables," which explicitly specify the choice of production during top-down parsing. These sets are also useful during bottom-up parsing.

3.4.1 recursive-descent parsing

```
    void A() {
1)      Choose an A-production, A → X1 X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

Figure: A typical procedure for a non-terminal in a top-down parser

A recursive-descent parsing program consists of a set of procedures, one for each nonterminal. Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string. Note that this pseudo code is nondeterministic because it begins by choosing the A-production to apply in a manner that is not specified.

General recursive-descent parsing may require backtracking, meaning it may need to scan over the input repeatedly. However, backtracking is rarely needed for parsing programming language constructs, so backtracking parsers are not commonly used. Even for tasks like natural language parsing, backtracking is not very efficient, and table-based methods like the dynamic programming algorithm or the Earley method are preferred.

To allow for backtracking, the code needs to be modified. First, we cannot choose a unique A-production at line (1), so we must try each of several productions in some order. Then, failure at line (7) does not indicate ultimate failure, but suggests only that we need to return to line (1) and try another A-production. Only if there are no more A-productions to try do we declare that an input error has been found. To try another A-production, we need to be able to reset the input pointer to where it was when we first reached line (1). Thus, a local variable is needed to store this input pointer for future use.

3.4.2 First and Follow

The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar G. During top-

down parsing, FIRST and FOLLOW allow us to choose which production to apply based on the next input symbol. During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.

The function FIRST(X), where X is any string of grammar symbols, is defined as the set of terminals that begin strings derived from X. If X => ε, then ε is also in FIRST(X).

For example,

if X => cY, then c is in FIRST(X).

For a preview of how FIRST can be used during predictive parsing, consider two A-productions A -> α | β, where FIRST(α) and FIRST(β) are disjoint sets. We can then choose between these A-productions by looking at the next input.

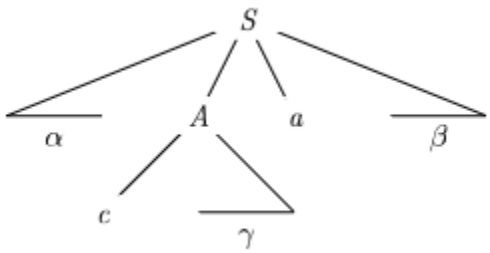


Figure Terminal c is in FIRST(A) and a is in FOLLOW (A)

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ε can be added to any FIRST set:

1. If X is a terminal, then FIRST(X) = {X}.
2. If X -> ε is a production or ε is in FIRST(Y) for all symbols Y in β, then add ε to FIRST(X).
3. If X -> Y1Y2...Yk is a production, then for i = 1 to k:
   - Add FIRST(Yi) - {ε} to FIRST(X).
   - If ε is not in FIRST(Yi), stop. Otherwise, continue to the next Yi.

To compute FOLLOW(A) for a nonterminal A, apply the following rules:

1. Add $ to FOLLOW(S), where S is the start symbol of the grammar, and $ is the special "endmarker" symbol.
2. For each production A -> αBβ, add FIRST(β) - {ε} to FOLLOW(B).
3. For each production A -> αB or A -> αBβ where FIRST(β) contains ε, add FOLLOW(A) to FOLLOW(B).

Repeat these rules until no more terminals can be added to any FOLLOW set.

3.5 BOTTOM-UP PARSING

A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top). It is convenient to describe parsing as the process of building parse trees, although a front end may in fact carry out a translation directly without building an explicit tree.

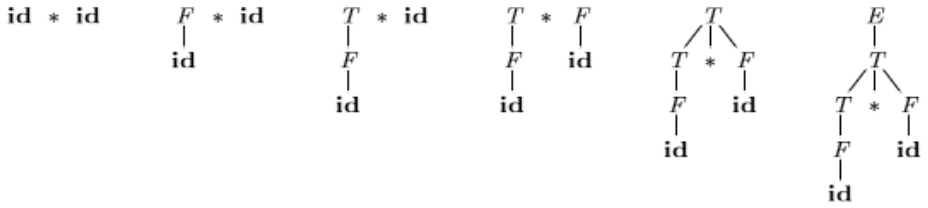


Figure: A bottom-up parse for id * id

Bottom-up parsing is a parsing technique that constructs a parse tree from leaves to root. It starts with the input tokens and uses a set of reduction rules to combine tokens into larger structures until the parse tree is complete. One common approach to bottom-up parsing is shift-reduce parsing, where the parser shifts input tokens onto a stack until it can reduce them to higher-level structures based on a predefined grammar. LR parsing is a type of shift-reduce parsing that is widely used in practice due to its efficiency and the availability of automated parser generators that can generate LR parsers from a given grammar.

3.5.1 Reductions

Bottom-up parsing involves the process of "reducing" a string w to the start symbol of the grammar. At each reduction step, a specific sub-string matching the body of a production is replaced by the non-terminal at the head of that production.

The key decisions during bottom-up parsing revolve around when to reduce and which production to apply as the parse proceeds.

For example, consider the following snapshots illustrating a sequence of reductions using the expression grammar. The reductions will be discussed in terms of the sequence of strings.

*id * id; F * id; T * id; T * F ; T ; E*

The strings in this sequence are formed from the roots of all the subtrees in the snapshots. The sequence starts with the input string "id * id." The first reduction produces "F * id" by reducing the leftmost "id" to "F," using the production F→id. The second reduction produces "T * id" by reducing "F" to "T."

Now, there is a choice between reducing the string "T," which is the body of E→T, and the string consisting of the second "id," which is the body of

F→id. Instead of reducing "T" to "E," the second "id" is reduced to "F," resulting in the string "T * F." This string then reduces to "T." The parse completes with the reduction of "T" to the start symbol E.

By definition, a reduction is the reverse of a step in a derivation (recall that in a derivation, a nonterminal in a sentential form is replaced by the body of one of its productions). The goal of bottom-up parsing is therefore to construct a derivation in reverse.

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

This derivation is in fact a rightmost derivation

3.5.2 Handle Pruning

Bottom-up parsing, during a left-to-right scan of the input, constructs a rightmost derivation in reverse. Informally, a "handle" is a substring that matches the body of a production, and its reduction represents one step along the reverse of a rightmost derivation.

For example, adding subscripts to the token's "id" for clarity, the handles during the parse of

"id1 * id2" according to the expression grammar are as follows:

1. Starting with "id1 * id2":
   - Handle: "id1" is reduced to "F" using the production F→id, resulting in "F * id2".
2. Continuing with "F * id2":
   - Handle: "F" is reduced to "T" using the production T→F, resulting in "T * id2".
3. Continuing with "T * id2":
   - Handle: "id2" is reduced to "F" using the production F→id, resulting in "T * F".
4. Finally, "T * F" is reduced to "T" using the production T→T∗F, resulting in the final parse tree.

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$T \to T * F$
T	T	$E \to T$

Note that although "T" is the body of the production E→T, the symbol "T" is not a handle in the sentential form "T * id2". If "T" were indeed replaced by "E", we would get the string "E * id2", which cannot be derived from the start symbol E. Thus, the leftmost substring that matches the body of some production need not be a handle.

Figure: Handles during a parse of id1 * id2

3.6 SYNTAX ANALYZER GENERATORS

A parser generator can be used to facilitate the construction of the front end of a compiler. We shall use the LALR parser generator Yacc as the basis of our discussion, and it is widely available. Yacc stands for "yet another compiler-compiler," reflecting the popularity of parser generators in the early 1970s when the first version of Yacc was created by S. C. Johnson. Yacc is available as a command on the UNIX system and has been used to help implement many production compilers.

3.6.1 Parser Generator YACC

A translator can be constructed using Yacc in the manner illustrated in Fig. First, a file, say translate.y, containing a Yacc specification of the translator is prepared. The UNIX system command yacc translate.y transforms the file translate.y into a C program called y.tab.c using the LALR method outlined below. The program y.tab.c is a representation of an LALR parser written in C, along with other C routines that the user may have prepared. The LALR parsing table is compacted. By compiling y.tab.c along with the ly library that contains the LR parsing program using the command cc y.tab.c -ly, we obtain the desired object program a.out that performs the translation specified by the original Yacc program. If other procedures are needed, they can be compiled or loaded with y.tab.c, just as with any C program.

A Yacc source program has three parts:

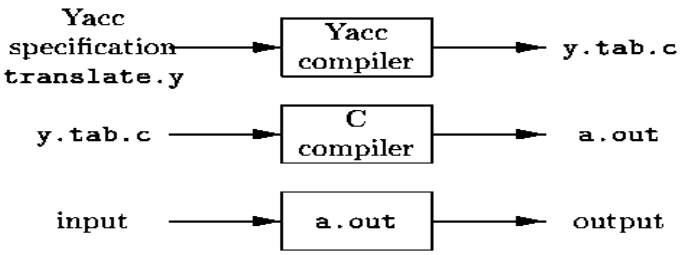


Figure: Creating an input/output translator with Yacc

declarations
%%
translation rules
%%
supporting C routines

Example : To illustrate how to prepare a Yacc source program, let us construct a simple desk calculator that reads an arithmetic expression, evaluates it, and then prints its numeric value. We shall build the desk calculator starting with the with the following grammar for arithmetic expressions:

E ! E + T j T

*T ! T * F j F*

F ! (E) j digit

The token digit is a single digit between 0 and 9.

The Declarations Part

There are two sections in the declarations part of a Yacc program; both are optional. In the rst section, we put ordinary C declarations, delimited by %{ and %}. Here we place declarations of any temporaries used by the translation rules or procedures of the second and third sections. Here, it contains only the include-statement

#include <ctype.h>

that causes the C pre-processor to include the standard header le <ctype.h>

that contains the predicate is digit.

Also, in the declarations part are declarations of grammar tokens

```
%token DIGIT
%{
#include<ctype.h>
%}
%token DIGIT
%%
line : expr '\n' { printf("%d\n", $1); }
;
expr : expr '+' term { $$ = $1 + $3; }
| term
;
term : term '*' factor { $$ = $1 * $3; }
| factor
;
factor : '(' expr ')' { $$ = $2; }
| DIGIT
;
%%
yylex() {
int c;
c = getchar();
if (isdigit(c)) {
yylval = c-'0';
return DIGIT;
}
return c;
}
```

declares DIGIT to be a token. Tokens declared in this section can then be used in the second and third parts of the Yacc specification. If Lex is used

to create the lexical analyzer that passes token to the Yacc parser, then these token declarations are also made available to the analyzer generated by Lex

3.6.2 The Translation Rules Part

In the part of the Yacc specification after the first %% pair, we put the translation rules. Each rule consists of a grammar production and the associated semantic action.

In a Yacc production, unquoted strings of letters and digits not declared to be tokens are taken to be non-terminals. A quoted single character, e.g., 'c', is taken to be the terminal symbol 'c', as well as the integer code for the token represented by that character (i.e., Lex would return the character code for 'c' to the parser, as an integer). Alternative bodies can be separated by a vertical bar, and a semicolon follows each head with its alternatives and their semantic actions. The first head is taken to be the start symbol.

A Yacc semantic action is a sequence of C statements. In a semantic action, the symbol $$ refers to the attribute value associated with the non-terminal of the head, while $i refers to the value associated with the i^{th} grammar symbol (terminal or non-terminal) of the body. The semantic action is performed whenever we reduce by the associated production, so normally the semantic action computes a value for $$ in terms of the $i's. In the Yacc specification, we have written the two E-productions.

E | E + T | T

and their associated semantic actions as:

```
expr : expr '+' term    { $$ = $1 + $3; }
     | term
     ;
```

In the Yacc specification, the nonterminal term in the first production is the third grammar symbol of the body, while + is the second. The semantic action associated with the first production adds the value of the expr and the term of the body and assigns the result as the value for the nonterminal expr of the head. We have omitted the semantic action for the second production altogether, since copying the value is the default action for productions with a single grammar symbol in the body. In general, { $$ = $1; } is the default semantic action.

Notice that we have added a new starting production:

line : expr '\n' { printf (" %d \n ", $1) ; }

This production says that an input to the desk calculator is to be an expression followed by a newline character. The semantic action associated with this production prints the decimal value of the expression followed by a newline character.

3.6.3 Using Yacc with Ambiguous Grammars

Let us now modify the Yacc specification so that the resulting desk calculator

becomes more useful. First, we shall allow the desk calculator to evaluate a

sequence of expressions, one to a line. We shall also allow blank lines between

expressions. We do so by changing the first rule to

```
lines : lines expr '\n'    { printf("%g\n", $2); }
      | lines '\n'
      | /* empty */
      ;
```

In Yacc, an empty alternative, as the third line is, denotes *. Second, we shall enlarge the class of expressions to include numbers with a decimal point instead of single digits and to include the arithmetic operators +, , (both binary and unary), *, and /. The easiest way to specify this class of expressions is to use the ambiguous grammar

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid - E \mid (E) \mid \textbf{number}$$

The grammar in the Yacc specification in above Fig. is ambiguous, which means that the LALR algorithm will generate parsing-action conflicts. Yacc reports the number of parsing-action conflicts that are generated. By invoking Yacc with a -v option, you can obtain a description of the sets of items and the parsing-action conflicts, as well as a readable representation of the LR parsing table showing how the conflicts were resolved.

By default, Yacc resolves parsing conflicts using two rules:

1. A reduce/reduce conflict is resolved by choosing the conflicting production listed first in the Yacc specification.
2. A shift/reduce conflict is resolved in favor of shift. This rule correctly resolves the shift/reduce conflict arising from the dangling-else ambiguity.

To customize the resolution of shift/reduce conflicts, Yacc provides a mechanism for assigning precedence's and associativity's to terminals in the declarations section. For example:

- %left '+' '-' makes + and - have the same precedence and be left-associative.
- %right '^' makes ^ right-associative.
- %nonassoc '<' makes < a non-associative binary operator.

Tokens are given precedences in the order in which they appear in the declarations part, lowest first. Tokens in the same declaration have the same precedence. Yacc resolves shift/reduce conflicts by comparing the precedence and associativity of the production and the terminal involved in the conflict. If the precedence of the production is greater than that of the terminal, or if they have the same precedence and the production is left-associative, then Yacc reduces; otherwise, it shifts.

You can also force a precedence to a production by appending %prec terminal to the production. This makes the precedence and associativity of the production the same as that of the terminal, which is defined in the declaration section. Yacc does not report shift/reduce conflicts resolved using this mechanism.

In your specific example,
%right UMINUS

assigns a higher precedence to the token
UMINUS

than that of * and /, and
%prec UMINUS

at the end of the production
expr : '-' expr

makes the unary-minus operator in this production have a higher precedence than any other operator

3.6.4 Error Recovery in YACC

In Yacc, error recovery is implemented using error productions. Here's how it works:

1. **Decision on Error Recovery**: The user decides which major nonterminals will have error recovery associated with them. These are typically nonterminals generating expressions, statements, blocks, and functions.

2. **Adding Error Productions**: The user adds error productions to the grammar of the form A ! error, where A is a major nonterminal and error is a Yacc reserved word. These error productions are treated as ordinary productions by Yacc.

3. **Error Handling**: When the parser encounters an error, it pops symbols from its stack until it finds the topmost state whose set of items includes an item of the form A ! * error. It then shifts a fictitious token error onto the stack, as if it had seen the token error in the input.

4. **Error Recovery Actions**:

    - If * is empty, a reduction to A occurs immediately, and the associated semantic action for A ! error is invoked. The parser then discards input symbols until it finds an input symbol on which normal parsing can proceed.

- If * is not empty, Yacc skips ahead on the input looking for a substring that can be reduced to *. If * consists entirely of terminals, it looks for this string of terminals on the input and reduces them by shifting them onto the stack. The parser will then reduce error to A and resume normal parsing.

For example, stmt ! Error ; specifies to the parser that it should skip just beyond the next semicolon on seeing an error and assume that a statement had been found. The semantic routine for this error production could generate a diagnostic message and set a flag to inhibit the generation of object code.

3.7 SUMMARY

The chapter discusses parsing techniques in compiler design, focusing on top-down and bottom-up parsing. It starts by explaining how parsers analyze the syntax of a program based on its grammar rules. Top-down parsing begins at the start symbol and tries to match the input string, while bottom-up parsing constructs a parse tree starting from the leaves and working towards the root.

The chapter introduces LL and LR parsing, which are common types of top-down and bottom-up parsing, respectively. LL parsing is predictive, meaning it looks ahead at the next input symbol to choose the correct production. LR parsing uses a more powerful shift-reduce technique to build a parse tree.

Error recovery is an important aspect of parsing, and Yacc provides mechanisms for handling errors in the input. Error productions can be added to the grammar to specify how the parser should recover from errors and continue parsing.

Overall, the chapter provides a comprehensive overview of parsing techniques, including their implementation and use in compiler construction.

3.8 REVIEW QUESTIONS

1. What is the key difference between top-down and bottom-up parsing techniques?
2. How does Yacc handle error recovery in parsing?
3. Explain the concept of handles in bottom-up parsing and their role in constructing a parse tree.

❄❄❄❄❄❄❄

4

SEMANTIC ANALYSIS

Unit Structure

4.0 Objective

4.1 Introduction

4.2 Role of Semantic Parser

4.3 Symbol Table Management

- 4.3.1 Symbol Tables
- 4.3.2 Multiple Symbol Tables
- 4.3.3 Efficient Imperative Symbol Tables
- 4.3.4 Efficient Functional Symbol Tables
- 4.3.5 Symbols in The Tiger Compiler

4.4 Type Checking and Type Systems.

- 4.4.1 Type-Checking Expressions
- 4.4.2 Type-Checking Variables, Subscripts, And Fields
- 4.4.3 Type-Checking Declarations

4.5 Attribute Grammars

- 4.5.1 Simplifications and Extensions to Attribute Grammars
- 4.5.2 Algorithms for Attribute Computation
- 4.5.3 The Dependence of Attribute Computation on The Syntax

4.6 Summary

4.7 Review Questions

4.0 OBJECTIVES

- Understand the role of a semantic parser in a compiler and its importance in translating the abstract syntax tree into executable code.
- Explore the concept of symbol tables and their management, including techniques for efficient storage and retrieval of symbols.
- Learn about type checking and type systems, including how expressions, variables, and declarations are type-checked to ensure program correctness.
- Study attribute grammars and their use in compiler design, including algorithms for attribute computation and their dependence on the syntax of the language being compiled.

4.1 INTRODUCTION

The semantic analysis phase of a compiler connects variable definitions to their uses, checks that each expression has a correct type, and translates the abstract syntax into a simpler representation suitable for generating machine code. This phase ensures that the program adheres to the rules of the programming language, such as type consistency, scope resolution, and the proper usage of identifiers. Semantic analysis typically involves creating and maintaining symbol tables that record information about variable names, function names, and their attributes. It also performs type checking to ensure that operations in the program are applied to compatible types. Additionally, this phase can include the generation of intermediate code or an abstract syntax tree that serves as a bridge between the high-level source code and the low-level machine code, facilitating further optimization and code generation phases.

4.2 ROLE OF SEMANTIC PARSER

Semantic analysis in a compiler serves two main purposes: ensuring the correctness of a program according to the rules of the programming language and enhancing the efficiency of the translated program. The extent of semantic analysis required varies significantly among different languages. For instance, dynamically-typed languages like LISP and Smalltalk might not require any static semantic analysis, while statically-typed languages such as Ada have stringent requirements for a program to be executable. Languages like Pascal and C fall somewhere in between these extremes, with Pascal being stricter than C but less so than Ada.

The first category of semantic analysis ensures the program adheres to language rules for proper execution. This includes tasks such as type checking, scope resolution, and ensuring variables are defined before use. The second category involves optimization techniques aimed at improving the execution efficiency of the translated program. Although these optimization methods are typically discussed under "code generation," the techniques for ensuring correctness also contribute to generating more efficient code. However, it's important to note that semantic analysis can only establish partial correctness of a program, not complete correctness, but it still significantly enhances the security and robustness of the program.

Implementing semantic analysis algorithms can be more complex than parsing algorithms due to the timing of the analysis during compilation. If semantic analysis is deferred until after syntactic analysis and the construction of an abstract syntax tree, the implementation becomes simpler, involving a traversal of the syntax tree with specific computations at each node. This approach is typical in multipass compilers. However, in single-pass compilers, where all operations, including code generation, must be performed in a single pass, the implementation becomes more ad hoc and complex. Fortunately, modern practices increasingly allow for multiple passes, simplifying the processes of semantic analysis and code generation.

Despite these challenges, studying attribute grammars and specification issues is valuable. It helps write clearer, more concise, and less error-prone code for semantic analysis, making the code easier to understand and maintain.

4.3 SYMBOL TABLE MANAGEMENT

4.3.1 Symbol Tables

This phase involves maintaining symbol tables (also known as environments) that map identifiers to their types and locations. As declarations of types, variables, and functions are processed, these identifiers are bound to specific meanings within the symbol tables. When identifiers are used (non-defining occurrences), they are looked up in the symbol tables. Each local variable in a program has a scope within which it is visible. For instance, in a Tiger expression let D in E end, all the variables, types, and functions declared in D are only visible until the end of E. As the semantic analysis reaches the end of each scope, the local identifier bindings are discarded.

An environment is a set of bindings denoted by the → arrow. For example, we could say that the environment σ_0 contains the bindings {g → string, a → int}, meaning the identifier a is an integer variable and g is a string variable.

Consider a simple example in the Tiger language:

tiger

```
1 function f(a:int, b:int, c:int) =
2 (print_int(a+c);
3 let var j := a+b
4 var a := "hello"
5 in print(a); print_int(j)
6 end;
7 print_int(b)
8 )
```

If we compile this program in the environment σ_0, the formal parameter declarations on line 1 extend the table to σ_1, which is σ_0 plus {a → int, b → int, c → int}. The identifiers in line 2 are looked up in σ_1. At line 3, the table σ_2 is created, which is σ_1 plus {j → int}. At line 4, σ_3 is created, which is σ_2 plus {a → string}.

How does the + operator for tables work when the two environments being "added" contain different bindings for the same symbol? For instance, when σ_2 and {a → string} map a to int and string, respectively? To ensure the scoping rules work as expected in real programming languages, {a →

string} should take precedence. Therefore, X + Y for tables is not the same as Y + X; bindings in the right-hand table override those in the left.

Finally, in line 6, σ_3 is discarded, and we revert to σ_1 for looking up the identifier b in line 7. At line 8, σ_1 is discarded, and we revert to σ_0.

How should this be implemented? There are two main approaches:

Functional Style: Keep σ_1 intact while creating σ_2 and σ_3. When σ_1 is needed again, it remains unchanged and ready to use.

Imperative Style: Modify σ_1 to become σ_2, which destructively updates σ_1. While σ_2 exists, σ_1 cannot be used. Once done with σ_2, the modification can be undone to restore σ_1. This involves a single global environment σ that transitions through σ_0, σ_1, σ_2, σ_3, σ_1, σ_0 at different times, along with an "undo stack" that tracks and reverses the updates. When a symbol is added to the environment, it is also added to the undo stack. At the end of a scope (e.g., line 6 or 8), symbols are popped from the undo stack, removing their latest bindings from σ and restoring their previous bindings.

```
structure M = struct
   structure E = struct
      val a = 5;
   end
   structure N = struct
      val b = 10
      val a = E.a + b
   end
   structure D = struct
      val d = E.a + N.a
   end
end
```

(a) An example in ML

```
package M;
class E {
    static int a = 5;
}
class N {
    static int b = 10;
    static int a = E.a + b;
}
class D {
    static int d = E.a + N.a;
}
```

(b) An example in Java

Both the functional and imperative styles of environment management can be used regardless of whether the language being compiled or the implementation language of the compiler is functional, imperative, or object-oriented.

4.3.2 Multiple Symbol Tables

In some languages, there can be multiple active environments simultaneously: each module, class, or record in the program has its own symbol table, σ. Let σ_0 be the base environment containing predefined functions, and let...

$$\sigma_1 = \{a \mapsto int\}$$
$$\sigma_2 = \{E \mapsto \sigma_1\}$$
$$\sigma_3 = \{b \mapsto int, a \mapsto int\}$$
$$\sigma_4 = \{N \mapsto \sigma_3\}$$
$$\sigma_5 = \{d \mapsto int\}$$
$$\sigma_6 = \{D \mapsto \sigma_5\}$$
$$\sigma_7 = \sigma_2 + \sigma_4 + \sigma_6$$

In ML, the module N is compiled using the environment σ0+σ2 to look up identifiers; D is compiled using σ0+σ2+σ4, resulting in {M→σ7}. In Java, forward reference is allowed (so inside N the expression D.d would be legal), thus E, N, and D are all compiled in the environment σ7; for this program, the result is still {M→σ7}.

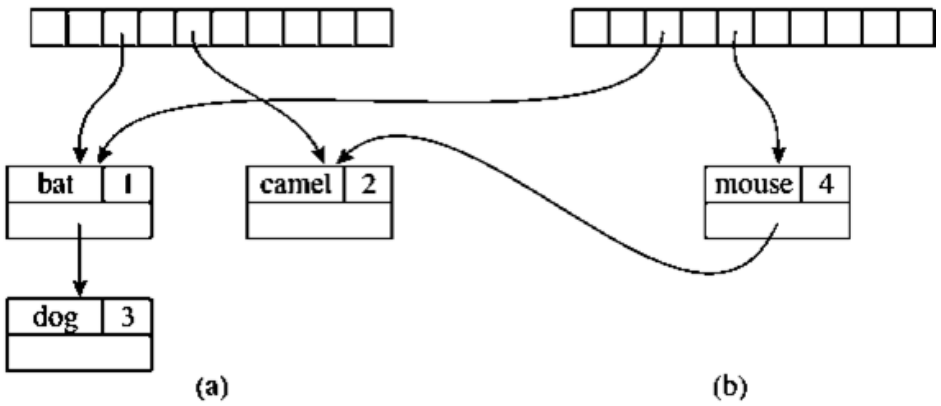
4.3.3 Efficient Imperative Symbol Tables

In programs with a large number of unique identifiers, efficient symbol tables are essential for quick lookup operations. Here is a sample implementation of a hash table using external chaining to manage collisions:

```
struct Bucket {
   string key;
   void *binding;
   struct Bucket *next;
};

#define SIZE 109
struct Bucket *table[SIZE];

unsigned int hash(char *str) {
   unsigned int hashVal = 0;
   char *s;
   for (s = str; *s; s++)
      hashVal = hashVal * 65599 + *s;
   return hashVal;
}

struct Bucket *createBucket(string key, void *binding, struct Bucket
*next) {
   struct Bucket *b = checked_malloc(sizeof(*b));
   b->key = key;
   b->binding = binding;
   b->next = next;
   return b;
}

void insert(string key, void *binding) {
   int index = hash(key) % SIZE;
   table[index] = createBucket(key, binding, table[index]);
}

void *lookup(string key) {
   int index = hash(key) % SIZE;
   struct Bucket *b;
   for (b = table[index]; b; b = b->next)
      if (strcmp(b->key, key) == 0) return b->binding;
   return NULL;
}
```

```
void pop(string key) {
   int index = hash(key) % SIZE;
   table[index] = table[index]->next;
}
```

This implementation uses a hash table with external chaining, making it efficient and supporting easy deletion operations. Each bucket in the hash table is a linked list of elements whose keys hash to the same index modulo SIZE.



Hash Tables

When adding a new binding to a key that already exists in the symbol table, the insert function in the provided hash table implementation leaves the existing binding in place and adds the new binding to the beginning of the linked list in the corresponding bucket. For example, if σ contains a ! → τ1 and a new binding a ! → τ2 is added, the table will contain both bindings, but a ! → τ2 will be the first in the list for key a.

Later, when pop(a) is called at the end of a's scope, only the topmost binding for a (the one added most recently) is removed. This is similar to how a stack operates, where elements are added and removed in a last-in-first-out (LIFO) manner. If pop(a) is called again, it will remove a ! → τ1, thus restoring the symbol table to its state before the addition of a ! → τ2.

4.3.4 Efficient Functional Symbol Tables

In the functional programming style, updating a symbol table is done by creating a new table that includes the new binding, rather than modifying the existing table. This approach ensures that the original table remains unchanged and available for further lookups. This concept is similar to adding numbers in arithmetic, where adding 7 and 8 results in a new value (15), but the original values (7 and 8) remain unchanged.

However, this non-destructive update approach is not efficient for hash tables. Adding a new binding to a hash table typically involves updating pointers in the table, which can be done quickly and efficiently. But this process destroys the previous mapping, making it unavailable for future

lookups. Another approach is to copy the entire array representing the hash table and then add the new element, but this is inefficient for large arrays because copying them for each new entry is costly.

Using binary search trees, on the other hand, allows for efficient functional additions. In a binary search tree, adding a new element involves creating a new node and adjusting the tree structure, but the original tree remains intact and available for further operations. This makes binary search trees suitable for functional updates without sacrificing efficiency.

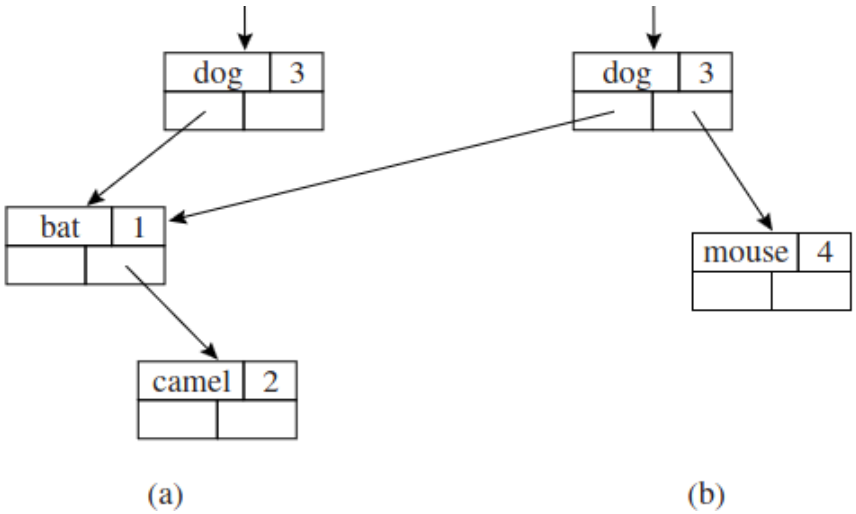


Figure: Binary Search Tables.

In a binary search tree representing the mapping m1 ={bat→1,camel→2,dog→3}, adding the binding mouse→4 to create the mapping m2 can be done efficiently without destroying m1. Adding a new node at depth d in the tree requires creating d new nodes, but the entire tree does not need to be copied. Therefore, creating a new tree that shares some structure with the old one can be done as efficiently as looking up an element, which takes O(logn) time for a balanced tree of n nodes. This approach demonstrates a persistent data structure, where a persistent red-black tree can be maintained to ensure logn access time while keeping the previous mappings intact.

4.3.5 Symbols in the Tiger Compiler

To efficiently handle strings in a hash table, we can convert each string to a symbol, enabling fast comparison and hashing. The Symbol module provides functions for creating symbols, accessing symbol names, and managing symbol tables. By using symbols, we can efficiently compare and hash strings without repeated string comparisons

```
/* symbol.h */
typedef struct S_symbol_ *S_symbol;
S_symbol S_Symbol(string);
string S_name(S_symbol);
typedef struct TAB_table_ *S_table;
S_table S_empty(void);
void S_enter(S_table t, S_symbol sym, void *value);
```

```
void *S_look(S_table t, S_symbol sym);
void S_beginScope(S_table t);
void S_endScope(S_table t);
```

PROGRAM *symbol.h, the interface for symbols and symbol tables.*

In the Tiger compiler, we use destructive-update environments, where each symbol is mapped to a binding. The S_empty() function creates a new symbol table, and S_beginScope() and S_endScope() manage the scope of the symbols. The S_beginScope() function remembers the current state of the table, and S_endScope() restores the table to the state before the most recent beginScope().

The symbol.c file implements the symbol table using a hash table. When a binding is entered, the corresponding symbol is hashed to an index, and a Binder object is placed at the head of the linked list for that index. If a previous binding exists for the same symbol, it is hidden by the new binding. The table.h file defines generic hash table operations for mapping keys to values.

Overall, the symbol module provides efficient handling of symbols and symbol tables, essential for compiler implementation, especially in managing scopes and mappings

```
#include <stdio.h>
#include <string.h>
#include "util.h"
#include "symbol.h"

/* Definition of the symbol structure */
struct S_symbol_ {
   string name;
   S_symbol next;
};

/* Helper function to create a new symbol */
static S_symbol mksymbol(string name, S_symbol next) {
   S_symbol s = checked_malloc(sizeof(*s));
   s->name = name;
   s->next = next;
   return s;
}
/* Size of the hash table */
#define SIZE 109

/* Hash function for strings */
static unsigned int hash(char *s0) {
   unsigned int h = 0;
   char *s;
   for(s = s0; *s; s++)
      h = h * 65599 + *s;
   return h;
}
```

```
/* Array to hold the hash table */
static S_symbol hashtable[SIZE];

/* Create a symbol from a string */
S_symbol S_Symbol(string name) {
   int index = hash(name) % SIZE;
   S_symbol syms = hashtable[index], sym;
   for(sym = syms; sym; sym = sym->next)
      if (0 == strcmp(sym->name, name))
         return sym;
   sym = mksymbol(name, syms);
   hashtable[index] = sym;
   return sym;
}

/* Get the name of a symbol */
string S_name(S_symbol sym) {
   return sym->name;
}

/* Create an empty symbol table */
S_table S_empty(void) {
   return TAB_empty();
}

/* Enter a symbol and its corresponding value into the table */
void S_enter(S_table t, S_symbol sym, void *value) {
   TAB_enter(t, sym, value);
}

/* Look up the value associated with a symbol in the table */
void *S_look(S_table t, S_symbol sym) {
   return TAB_look(t, sym);
}

/* Begin a new scope in the symbol table */
void S_beginScope(S_table t) {
   S_enter(t, &marksym, NULL);
}
/* End the current scope in the symbol table */
void S_endScope(S_table t) {
   S_symbol s;
   do
      s = TAB_pop(t);
   while (s != &marksym);
}
```

PROGRAM *Symbol table (symbol.c) implementation*

/ table.h - generic hash table */*

```
/* Opaque type representing a hash table */
typedef struct TAB_table_ *TAB_table;

/* Create a new empty hash table */
TAB_table TAB_empty(void);

/* Enter a key-value pair into the hash table */
void TAB_enter(TAB_table t, void *key, void *value);

/* Look up the value associated with a key in the hash table */
void *TAB_look(TAB_table t, void *key);

/* Pop the most recent binding from the hash table and return its key */
void *TAB_pop(TAB_table t);
```

When a new binding x ! → b is entered using *S_enter(table, x, b),* the key x is hashed to an index *i,* and a Binder object *x ! → b* is placed at the head of the linked list for the ith bucket. If the table already contained a binding *x ! → b′,* that previous binding would still be in the bucket, hidden by the new binding *x ! → b.* This allows for the implementation of undo operations like *beginScope* and *endScope.*

The key x is not a character string, but rather the S_symbol pointer itself. The table module implements generic pointer hash tables (TAB_table), mapping a key type (void*) to a binding type (also void*).

To avoid potential programming mistakes due to the use of void*, the symbol module encapsulates these operations with functions like S_empty, S_enter, etc., where the key type is S_symbol instead of void*.

Additionally, an auxiliary stack is used to keep track of the order in which symbols were "pushed" into the symbol table. When a new binding x ! → b is entered, x is pushed onto this stack. The beginScope operation pushes a special marker onto the stack. To implement endScope, symbols are popped off the stack down to and including the topmost marker. As each symbol is popped, the head binding in its bucket is removed.

The auxiliary stack can be integrated into the Binder by using a global variable top that shows the most recent symbol bound in the table. "Pushing" is achieved by copying top into the prevtop field of the Binder, thus threading the "stack" through the binders

4.4 TYPE CHECKING AND TYPE SYSTEMS

4.4.1 Type-Checking Expressions

The Semant module (semant.h, semant.c) performs semantic analysis – including type-checking – of abstract syntax. It contains four functions that recur over syntax trees:

```
struct expty transVar   (S_table venv,   S_table tenv, A_var v);
struct expty transExp   (S_table venv,   S_table tenv, A_exp a);
```

```
void transDec      (S_table venv,     S_table tenv, A_dec d);
struct Ty_ty transTy     (                    S_table tenv, A_ty a);
```

The type-checker is a recursive function of the abstract syntax tree. I will call it transExp because we will later augment this function not only to type-check but also to translate the expressions into intermediate code. The arguments of transExp are a value environment venv, a type environment tenv, and an expression. The result will be an expty, containing a translated expression and its Tiger-language type:

```
struct expty {Tr_exp exp; Ty_ty ty;};
struct expty expTy(Tr_exp exp, Ty_ty ty) {
struct expty e; e.exp=exp; e.ty=ty; return e;
}
```

To avoid discussing intermediate code, we'll define a dummy Translate module as follows:

*typedef void *Tr_exp;* and we'll use NULL for every value.

Let's consider a simple case: an addition expression e1 + e2. In Tiger, both operands must be integers, which the type-checker must verify. The result of the addition will also be an integer, as determined by the type-checker.

In many languages, addition is overloaded, meaning the + operator can represent both integer addition and real (floating-point) addition. If both operands are integers, the result is an integer. If both operands are real numbers, the result is real. However, if one operand is an integer and the other is a real number, the integer is typically implicitly converted to a real number, and the result is also a real number. This conversion is usually made explicit in the machine code generated by the compiler.

Implementing Tiger's non-overloaded type-checking for addition is straightforward.:

```
struct expty transExp(S_table venv, S_table tenv, A_exp a) {
   switch (a->kind) {
      // Other cases for different kinds of expressions

      case A_opExp: {
         A_oper oper = a->u.op.oper;
         struct expty left = transExp(venv, tenv, a->u.op.left);
         struct expty right = transExp(venv, tenv, a->u.op.right);
         if (oper == A_plusOp) {
            if (left.ty->kind != Ty_int)
               EM_error(a->u.op.left->pos, "integer required");
            if (right.ty->kind != Ty_int)
               EM_error(a->u.op.right->pos, "integer required");
            return expTy(NULL, Ty_Int());
         }
         // Handle other operators similarly
      }
```

```
        // Handle other cases for different kinds of expressions

        default:
            // Handle other cases
            assert(0); // should have returned from some clause of the switch
    }
}
```

The code snippet provided is a part of the function transExp which translates expressions in the Tiger programming language. The switch statement is used to handle different kinds of expressions (A_exp). Inside the case A_opExp, it checks if the operator is A_plusOp, which represents addition. It then recursively calls transExp for the left and right operands of the addition.

If the left operand is not an integer, it generates an error message using EM_error. Similarly, if the right operand is not an integer, it also generates an error message. Finally, if both operands are integers, it returns an expTy structure with a Ty_Int type.

The assert(0) statement is a safety measure to ensure that the function always returns a value. If the function somehow reaches this point, it indicates a logic error because it should have returned from one of the case clauses earlier in the function

4.4.2 Type-Checking Variables, Subscripts, and Fields

The transVar function recursively processes A_var expressions in a manner similar to transExp for A_exp expressions.

```
struct expty transVar(S_table venv, S_table tenv, A_var v) {
    switch(v->kind) {
        case A_simpleVar: {
            E_enventry x = S_look(venv, v->u.simple);
            if (x && x->kind == E_varEntry)
                return expTy(NULL, actual_ty(x->u.var.ty));
            else {
                EM_error(v->pos, "undefined variable %s", S_name(v-
>u.simple));
                return expTy(NULL, Ty_Int());
            }
        }
        case A_fieldVar:
        // Handle fieldVar case
        ...
    }
    assert(0); /* should have returned from some clause of the switch */
}
```

The transVar function verifies SimpleVar expressions by checking the environment for the variable's binding. If the identifier is found and is bound to a VarEntry (not a FunEntry), its type is extracted from the VarEntry.

For function calls, the function identifier is looked up in the environment, yielding a FunEntry containing a list of parameter types. These types are then matched against the arguments in the function call expression. The FunEntry also provides the result type of the function, which becomes the type of the function call.

Each kind of expression has its own type-checking rules, but those not yet described follow similar patterns of environment lookup and type matching.

4.4.3 Type-Checking Declarations

Environments are managed and updated by declarations in Tiger, with declarations appearing exclusively within a let expression. Type-checking a let expression involves using transDec to translate declarations:

```
struct expty transExp(S_table venv, S_table tenv, A_exp a) {
   switch(a->kind) {
      ...
      case A_letExp: {
         struct expty exp;
         A_decList d;
         S_beginScope(venv);
         S_beginScope(tenv);
         for (d = a->u.let.decs; d; d = d->tail)
            transDec(venv, tenv, d->head);
         exp = transExp(venv, tenv, a->u.let.body);
         S_endScope(tenv);
         S_endScope(venv);
         return exp;
      }
      ...
   }
}
```

In this excerpt, transExp sets the current state of the environments using beginScope(), iterates over the declaration list a->u.let.decs to update the environments venv and tenv with new declarations, translates the body expression a->u.let.body, and then reverts the environments to their original state using endScope().

Variable Declarations

Processing a declaration in Tiger involves augmenting an environment with a new binding, which is then used in subsequent declarations and expressions. For instance, processing a variable declaration without a type constraint, such as var x := exp, is straightforward:

```
void transDec(S_table venv, S_table tenv, A_dec d) {
```

```
  switch(d->kind) {
    case A_varDec: {
      struct expty e = transExp(venv, tenv, d->u.var.init);
      S_enter(venv, d->u.var.var, E_VarEntry(e.ty));
    }
    ...
  }
}
```

If a type constraint is present, as in var x : type-id := exp, compatibility between the constraint and the initializing expression must be checked. Additionally, initializing expressions of type Ty_Nil must be constrained by a Ty_Record type.

Type Declarations

Nonrecursive type declarations are relatively straightforward:

```
void transDec(S_table venv, S_table tenv, A_dec d) {
  ...
  case A_typeDec: {
    S_enter(tenv, d->u.type->head->name, transTy(d->u.type->head->ty));
  }
}
```

The transTy function translates type expressions from the abstract syntax (A_ty) to digested type descriptions (Ty_ty). This translation involves recursively traversing the structure of an A_ty, converting A_recordTy into Ty_Record, and so on. During translation, transTy looks up any symbols it finds in the type environment tenv.

Function Declarations

Function declarations are more complex:

```
void transDec(S_table venv, S_table tenv, A_dec d) {
  switch(d->kind) {
    ...
    case A_functionDec: {
      A_fundec f = d->u.function->head;
      Ty_ty resultTy = S_look(tenv, f->result);
      Ty_tyList formalTys = makeFormalTyList(tenv, f->params);
      S_enter(venv, f->name, E_FunEntry(formalTys, resultTy));
      S_beginScope(venv);
      {
        A_fieldList l;
        Ty_tyList t;
        for(l = f->params, t = formalTys; l; l = l->tail, t = t->tail)
          S_enter(venv, l->head->name, E_VarEntry(t->head));
```

```
            }
            transExp(venv, tenv, d->u.function->body);
            S_endScope(venv);
            break;
        }
    }
}
```

This implementation is simplified and handles only single functions with a result, not handling recursive functions or errors like undeclared type identifiers. It constructs a FunEntry for the function, enters formal parameters into the value environment, processes the body, and then discards the formal parameters from the environment.

Recursive Declarations

For mutually recursive types or functions, headers are first entered into the environment and then bodies are processed using these headers. Headers for types are entered as Ty_Name types with empty bindings:

```
S_enter(tenv, name, Ty_Name(name, NULL));
```

Subsequently, transTy stops at Ty_Name types to prevent errors when looking up types. Illegal cycles in mutually recursive type declarations should be detected by the type-checker. Mutually recursive functions are handled similarly, gathering information about headers in the first pass and processing bodies in the second pass.

4.5 ATTRIBUTE GRAMMARS

In syntax-directed semantics, attributes are associated directly with the grammar symbols of the language, including terminals and nonterminals. If X is a grammar symbol and a is an attribute associated with X, we denote the value of an associated with X as X.a. This notation is akin to a record field designator in Pascal or a structure member operation in C. Typically, attributes are calculated and stored in the nodes of a syntax tree using record fields or structure members.

For a collection of attributes a1, ..., an, the principle of syntax-directed semantics states that for each grammar rule $X_0 \rightarrow X_1X_2...X_n$ (where X_0 is a nonterminal and the other X_i are arbitrary symbols), the values of the attributes $X_i.a$ are related to the values of the attributes of the other symbols in the rule. If the same symbol X_i appears multiple times in the rule, each occurrence must be distinguished from the others by suitable subscripting to differentiate their attribute values.

Each relationship is specified by an attribute equation or semantic rule of the form $X_i.a = f_{ij}(X_0.a, X_1.a, ..., X_k.a)$, where f_{ij} is a mathematical function of its arguments. An attribute grammar for the attributes a1, ..., an consists of all such equations for all the grammar rules of the language.

Grammar Rule	Semantic Rules
Rule 1	Associated attribute equations
.	.
.	.
.	.
Rule *n*	Associated attribute equations

We proceed immediately to several examples.

While attribute grammars can seem complex, the functions f_{ij} are usually quite simple in practice. Attributes typically depend on only a few other attributes, allowing them to be separated into small, independent sets of interdependent attributes, and attribute grammars can be written separately for each set.

Attribute grammars are often presented in tabular form, with each grammar rule listed along with the set of attribute equations or semantic rules associated with that rule.

The given grammar describes the syntax of unsigned numbers, where each number is composed of digits (0-9). The grammar has two nonterminals: number and digit. Each number has an attribute val representing its numerical value.

1. For digits, the value is directly computable from the digit itself. For example, the attribute equation digit.val = 0 represents the value of digit 0.

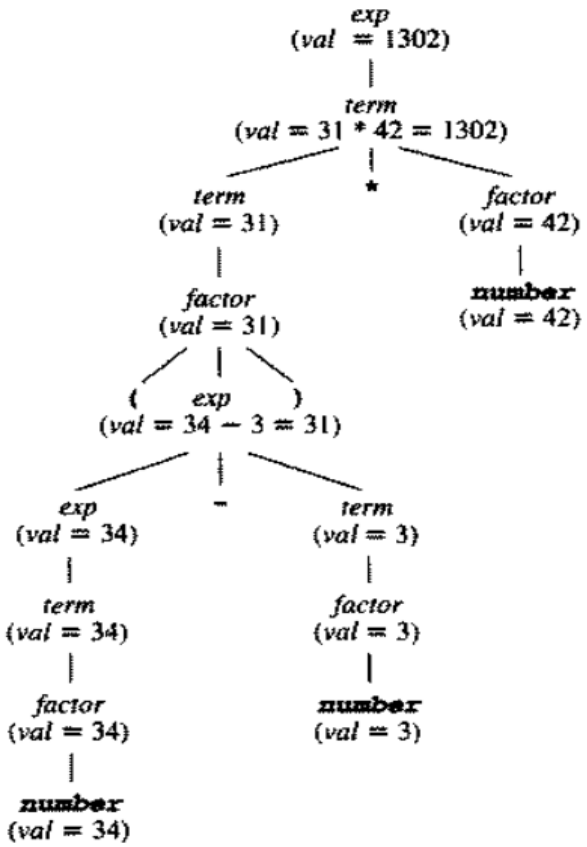
2. For numbers, if a number consists of a single digit, its value is simply the value of that digit. This is represented by the equation number.val = digit.val.

3. If a number consists of more than one digit, its value is computed by shifting the value of the leftmost digit one decimal place to the left and adding the value of the rightmost digit. For example, for the number 34, the value is calculated as 3 * 10 + 4. This is represented by the equation number.val = number2.val * 10 + digit.val, where number2 represents the leftmost digit.

These equations define the relationship between the syntax of numbers and their semantic value. They are used to compute the value of a number during semantic analysis, typically by traversing a parse tree of the expression.

The attribute grammar for the val attribute, which shows how the value of a number is computed based on its digits.

In summary, the attribute grammar provides a way to compute the numerical value of numbers based on their syntax, enabling semantic analysis to be performed on arithmetic expressions.



The given attribute grammar defines a dtype attribute for variable declarations in a C-like syntax, where variables can be of type int or float. The dtype attribute represents the data type of the variables. Here's a summary of the attribute grammar:

1. The dtype attribute of the nonterminal type is determined by the token it represents (int or float), corresponding to the set {integer, real}.

2. For variable lists (var-list), each identifier (id) in the list has the same dtype as the entire list, as per the equations associated with var-list.

3. The dtype of the entire declaration (decl) is the dtype of the var-list, as per the equation associated with the grammar rule for decl.

4. There is no equation involving the dtype of the nonterminal decl, indicating that a declaration need not have a dtype specified.

The attribute equations can be applied to a parse tree to compute the dtype attribute for each identifier in a variable declaration.

In cases where the grammar allows syntactically correct but semantically erroneous combinations (e.g., 1890), an additional error value is needed to handle such cases. This can be done using conditional expressions in the attribute equations to handle error conditions appropriately.

Overall, the attribute grammar provides a way to determine the data type of variables in a C-like syntax, enabling semantic analysis of variable declarations.

4.5.1 Simplifications and Extensions to Attribute Grammars

The use of an if-then-else expression in attribute equations expands the range of expressions that can be used, allowing for more flexibility in defining attribute values. This enhances the metalanguage for attribute grammars, which is the set of expressions allowed in attribute equations. A clear and expressive metalanguage is essential to avoid confusion and to facilitate the translation of attribute equations into working code for a semantic analyzer.

In addition to arithmetic and logical expressions, the metalanguage may include other types of expressions, such as if-then-else expressions, and occasionally case or switch expressions. These features make the metalanguage closely resemble an actual programming language, which is beneficial when translating attribute equations into executable code.

Another useful feature is the ability to use functions in attribute equations. Functions like numval(D) can be used to simplify attribute equations, especially when dealing with multiple similar cases. The definition of these functions needs to be provided separately, but they can significantly improve the readability and conciseness of attribute equations.

It's also mentioned that an ambiguous, but simpler, form of the original grammar can be used in attribute grammars, as long as the ambiguity has been resolved by the parser. This allows for more straightforward attribute definitions, without introducing ambiguity in the resulting attributes.

4.5.2 Algorithms for Attribute Computation

To implement an attribute grammar in a compiler, the attribute equations are translated into computation rules. Each attribute equation assigns the value of a functional expression on the right-hand side to the attribute on the left-hand side. For this assignment to succeed, the values of all attributes used in the expression must already exist.

In the specification of attribute grammars, the order in which the equations are written doesn't affect their validity, but in implementation, an order for evaluating and assigning attributes must be determined. This order is constrained by the dependencies between attributes, which are made explicit using dependency graphs. Dependency graphs represent the order constraints on attribute computation.

Each grammar rule choice in an attribute grammar has an associated dependency graph. The graph has a node for each attribute of each symbol in the rule, and there is an edge from each attribute on the right-hand side of an equation to the attribute on the left-hand side, representing the dependency. The dependency graph for a legal string in the language is the union of the dependency graphs for each grammar rule choice along the parse tree of the string.

When drawing dependency graphs, nodes for each symbol are grouped together to reflect the structured dependencies around a parse tree. For example, in the attribute grammar for numbers, each symbol has a single

node for its "val" attribute. The dependency graph for the grammar rule "number → number, digit" reflects the dependency of "number.val" on "number2.val" and "digit.val" in the equation "number.val = number2.val * 10 + digit.val"

4.5.3 The Dependence of Attribute Computation on the Syntax

The properties of attributes in an attribute grammar are closely tied to the structure of the grammar itself. Changes to the grammar that don't affect the legal strings of the language can significantly impact the computation of attributes, making it either simpler or more complex.

For example, let's consider the grammar for simple declarations:

decl → type var-list
type → int float
var-list → id, var-list | id

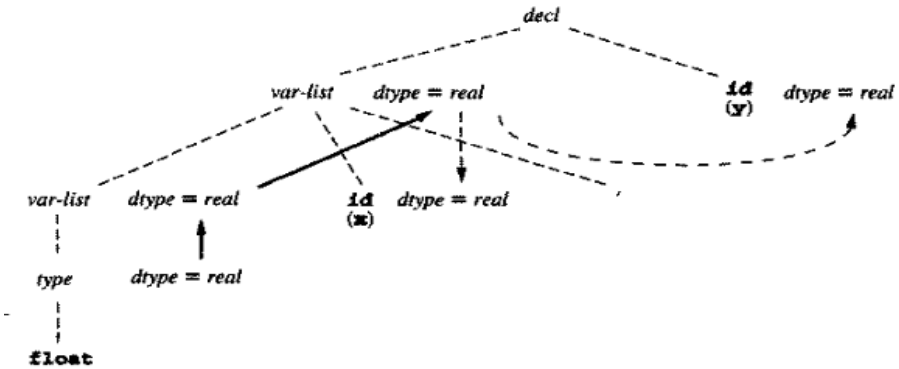
In this grammar, the dtype attribute is inherited. However, if we modify the grammar slightly as follows:

decl → var-list id
var-list → var-list id, | type
type → int | float

The language accepted by the grammar remains the same, but now the dtype attribute becomes synthesized. The corresponding attribute grammar is as follows:

Grammar Rule	Semantic Rules
decl → var-list id	id.dtype = var-list.dtype
var-list → var-list id, | type	var-list.dtype = var-list2.dtype | type.dtype
type → int | float	type.dtype = integer | real

This change affects how the dtype attribute is computed. In the new grammar, dtype is computed bottom-up (synthesized) rather than top-down (inherited). The parse tree for the string "float x, y" with attribute values and dependencies is shown in Figure below. Note that while it may appear that there are inherited attributes in the figure, these dependencies are actually to leaves in the parse tree and can be achieved by operations at the appropriate parent nodes, so they are not considered true inheritances



Indeed, while it is theoretically possible to modify a grammar to change inherited attributes into synthesized attributes, this approach can often lead to more complex and less understandable grammars and semantic rules. As a result, it is generally not recommended as a solution for computing inherited attributes.

However, if the computation of attributes in a grammar becomes overly complex or difficult, it may indicate that the grammar itself is not well-suited for attribute computation. In such cases, it may be worthwhile to consider modifying the grammar to make the attribute computation more straightforward and manageable.

4.6 SUMMARY

The chapter discusses attribute grammars, which associate attributes with grammar symbols and use rules to compute these attributes. It explains how attributes are related to the syntax of a language and how they can be used to derive meaning from the structure of the language. It introduces the concept of a metalanguage for writing attribute equations, which includes arithmetic, logical expressions, and if-then-else statements. The chapter also discusses the use of functions in attribute equations and how they can simplify the specification of attributes.

Dependency graphs are introduced as a way to represent the dependencies between attributes in a grammar. These graphs help determine the order in which attributes should be computed to ensure that all dependencies are met. The chapter also discusses how modifications to a grammar can affect the computation of attributes. While it is possible to change a grammar to turn inherited attributes into synthesized attributes, this can often lead to more complex grammars. It suggests that if attribute computation becomes overly complex, it may be a sign that the grammar needs to be revised for better attribute computation.

4.7 REVIEW QUESTIONS

1. How are attributes related to the syntax of a language, and how are they used to derive meaning from the language's structure?
2. What is a metalanguage in the context of attribute grammars, and what types of expressions are typically allowed in a metalanguage?
3. How are dependency graphs used in attribute grammars, and what role do they play in determining the order of attribute computation?

❄❄❄❄❄❄❄

5

INTERMEDIATE CODE GENERATION

Unit Structure

5.0 Objective

5.1 Intermediate representations (IR)

5.2 Three-address code generation

5.3 Quadruples and triples

5.4 Syntax-directed translation

5.5 Summary

5.6 Exercise

5.7 References

5.0 OBJECTIVE

- To explore the Concept of Intermediate Code Generation
- To understand the concept of Three Address Code.
- To understand different types of representation of Strings.

5.1 INTERMEDIATE REPRESENTATIONS (IR)

In the analysis-synthesis model of a compiler, the front end translates a source program into an intermediate code that is independent of the machine, and the back end uses this intermediate code to generate the target code, which can be understood by the machine. The benefits of using machine-independent intermediate code include:

- **Enhanced Portability:** Machine-independent intermediate code significantly improves the portability of the compiler. Without intermediate code, the compiler would need to translate the source language directly to the target machine language, requiring a full native compiler for each new machine. This necessitates modifications in the compiler according to the specific machine specifications.

- **Facilitated Retargeting:** Intermediate code allows for easier adaptation of the compiler to different target machines. Instead of rewriting the entire compiler, only the back end needs to be adjusted to accommodate the new machine architecture.

- **Improved Optimization:** Source code optimization becomes more manageable by modifying the intermediate code. This allows for better performance improvements in the source code.

The parser, a crucial component of the compiler's front end, uses a Context-Free Grammar (CFG) to validate the input string and produce output for the subsequent phase. The output can be either a parse tree or an abstract syntax tree. To interleave semantic analysis with the syntax analysis phase of the compiler, Syntax Directed Translation is employed. This approach integrates semantic analysis directly into the parsing process, ensuring that the semantic meaning of the code is analyzed as the syntax is being processed.

Definition

Syntax Directed Translation (SDT) enhances grammar rules to facilitate semantic analysis. It involves passing information through the parse tree in the form of attributes attached to the nodes, which can be done in a bottom-up or top-down manner. SDT rules use:

- Lexical values of nodes
- Constants
- Attributes associated with the non-terminals in their definitions

The general approach to Syntax Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes by visiting them in a specific order. Often, this translation can be accomplished during parsing without the need to build an explicit tree. This allows for efficient and integrated semantic analysis during the parsing process.

```
E -> E+T | T
T -> T*F | F
F -> INTLIT
```

This is a grammar to syntactically validate an expression having additions and multiplications in it.

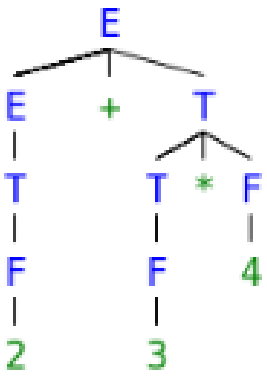
```
E -> E+T     { E.val = E.val + T.val }   PR#1
E -> T       { E.val = T.val }           PR#2
T -> T*F     { T.val = T.val * F.val }   PR#3
T -> F       { T.val = F.val }           PR#4
F -> INTLIT  { F.val = INTLIT.lexval }   PR#5
```

To understand translation rules further, consider the Syntax Directed Translation (SDT) augmented to the production rule [E -> E + T]. In this context, the attribute val is associated with both non-terminals E and T. The right-hand side of the translation rule corresponds to the attribute values of the right-side nodes of the production rule, and vice-versa.

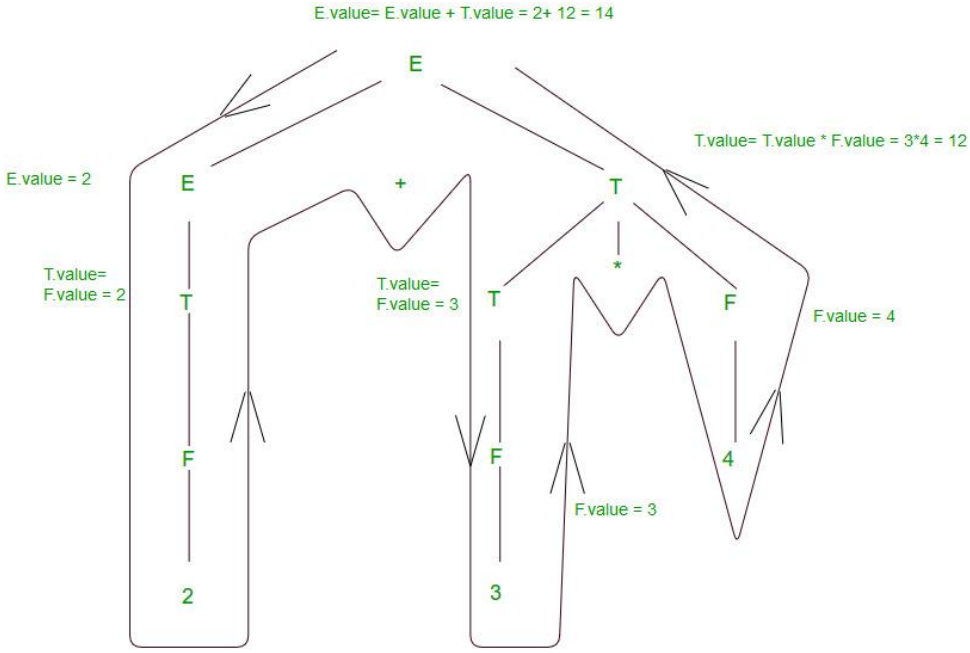
Generalizing, SDT involves augmenting rules to a Context-Free Grammar (CFG) by associating:

- A set of attributes to every node of the grammar.

- A set of translation rules to every production rule, using attributes, constants, and lexical values.



To evaluate translation rules, we can employ a depth-first search (DFS) traversal on the parse tree. This is feasible because SDT rules do not impose a specific order on evaluation, provided that children's attributes are computed before their parents' attributes in grammars where all attributes are synthesized. Otherwise, we would need to determine the best traversal strategy to evaluate all attributes in one or more passes through the parse tree.



The diagram above illustrates how semantic analysis occurs. The flow of information happens bottom-up, with all children's attributes computed before their parents' attributes, as discussed. Right-hand side nodes are sometimes annotated with a subscript to distinguish between children and parents.

Synthesized Attributes are attributes that depend only on the attribute values of children nodes. For example, in the production rule [E -> E + T { E.val = E.val + T.val }], the attribute val of node E is synthesized. If all the semantic attributes in an augmented grammar are synthesized, a single

depth-first search traversal in any order is sufficient for the semantic analysis phase.

Inherited Attributes are attributes that depend on the attributes of parents and/or siblings. For example, in the production rule [Ep -> E + T { Ep.val = E.val + T.val, T.val = Ep.val }], where E and Ep are the same production symbols annotated to differentiate between parent and child, val is an inherited attribute corresponding to node T.

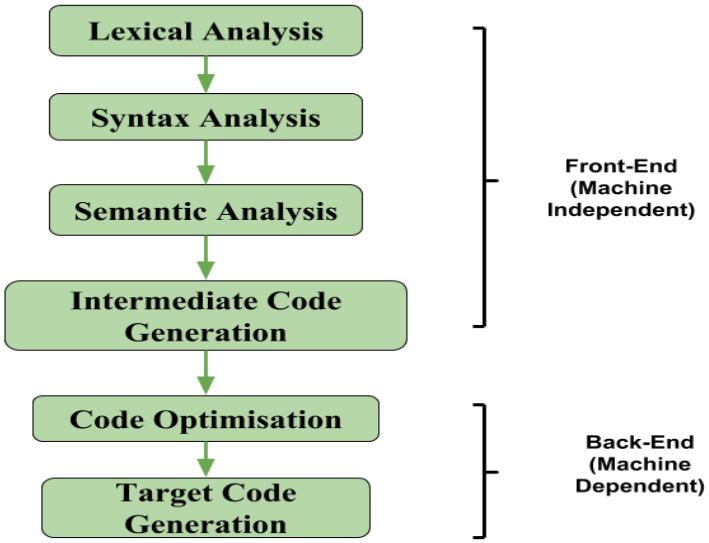
Thus, the flow of semantic analysis for synthesized attributes follows a straightforward bottom-up traversal, while inherited attributes require careful consideration of parent and sibling nodes to compute the attribute values correctly.

Advantages of Syntax Directed Translation:

- **Ease of implementation:** SDT is a simple and easy-to-implement method for translating a programming language. It provides a clear and structured way to specify translation rules using grammar rules.
- **Separation of concerns:** SDT separates the translation process from the parsing process, making it easier to modify and maintain the compiler. It also separates the translation concerns from the parsing concerns, allowing for more modular and extensible compiler designs.
- **Efficient code generation:** SDT enables the generation of efficient code by optimizing the translation process. It allows for the use of techniques such as intermediate code generation and code optimization.

Disadvantages of Syntax Directed Translation:

- **Limited expressiveness:** SDT has limited expressiveness in comparison to other translation methods, such as attribute grammars. This limits the types of translations that can be performed using SDT.
- **Inflexibility:** SDT can be inflexible in situations where the translation rules are complex and cannot be easily expressed using grammar rules.
- **Limited error recovery:** SDT is limited in its ability to recover from errors during the translation process. This can result in poor error messages and may make it difficult to locate and fix errors in the input program.



If we generate machine code directly from source code then for n target machine we will have optimizers and n code generator but if we will have a machine-independent intermediate code, we will have only one optimizer. Intermediate code can be either language-specific (e.g., Bytecode for Java) or language. independent (three-address code). The following are commonly used intermediate code representations:

1. Postfix Notation:

- Also known as reverse Polish notation or suffix notation.
- In the infix notation, the operator is placed between operands, e.g., *a + b*. Postfix notation positions the operator at the right end, as in *ab +*.
- For any postfix expressions *e1* and *e2* with a binary operator *(+)* , applying the operator yields *e1e2+*.
- Postfix notation eliminates the need for parentheses, as the operator's position and arity allow unambiguous expression decoding.
- In postfix notation, the operator consistently follows the operand.

  Example 1: The postfix representation of the expression (a + b) * c is : ab + c *

  Example 2: The postfix representation of the expression (a – b) * (c + d) + (a – b) is : ab – cd + *ab -+

2. Three-Address Code:

- A three address statement involves a maximum of three references, consisting of two for operands and one for the result.
- A sequence of three address statements collectively forms a three address code.

- The typical form of a three address statement is expressed as *x = y op z*, where *x, y*, and *z* represent memory addresses.
- Each variable *(x, y, z)* in a three address statement is associated with a specific memory location.
- While a standard three address statement includes three references, there are instances where a statement may contain fewer than three references, yet it is still categorized as a three address statement.

  Example: The three address code for the expression a + b * c + d : T1 = b * c T2 = a + T1 T3 = T2 + d; T 1 , T2 , T3 are temporary variables.

There are 3 ways to represent a Three-Address Code in compiler design:

i) Quadruples

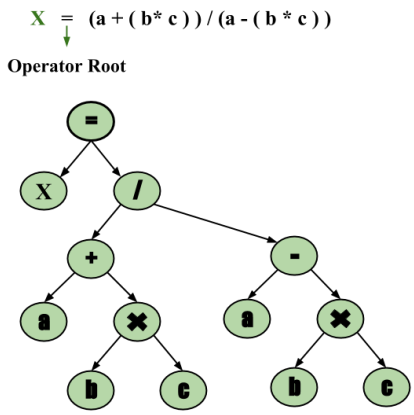
ii) Triples

iii) Indirect Triples

3. Syntax Tree:

- A syntax tree serves as a condensed representation of a parse tree.
- The operator and keyword nodes present in the parse tree undergo a relocation process to become part of their respective parent nodes in the syntax tree. the internal nodes are operators and child nodes are operands.
- Creating a syntax tree involves strategically placing parentheses within the expression. This technique contributes to a more intuitive representation, making it easier to discern the sequence in which operands should be processed.
- The syntax tree not only condenses the parse tree but also offers an improved visual representation of the program's syntactic structure,

  Example: x = (a + b * c) / (a – b * c)



Advantages of Intermediate Code Generation:

- **Easier to implement:** Intermediate code generation can simplify the code generation process by reducing the complexity of the input code, making it easier to implement.
- **Facilitates code optimization:** Intermediate code generation can enable the use of various code optimization techniques, leading to improved performance and efficiency of the generated code.
- **Platform independence:** Intermediate code is platform-independent, meaning that it can be translated into machine code or bytecode for any platform.
- **Code reuse:** Intermediate code can be reused in the future to generate code for other platforms or languages.
- **Easier debugging:** Intermediate code can be easier to debug than machine code or bytecode, as it is closer to the original source code.

Disadvantages of Intermediate Code Generation:

- **Increased compilation time:** Intermediate code generation can significantly increase the compilation time, making it less suitable for real-time or time-critical applications.
- **Additional memory usage:** Intermediate code generation requires additional memory to store the intermediate representation, which can be a concern for memory-limited systems.
- **Increased complexity:** Intermediate code generation can increase the complexity of the compiler design, making it harder to implement and maintain.
- **Reduced performance:** The process of generating intermediate code can result in code that executes slower than code generated directly from the source code.

5.2 THREE ADDRESS CODE IN COMPILER :-

Three-address code is a type of intermediate code that is easy to generate and can be easily converted to machine code. It uses at most three addresses and one operator to represent an expression, with the computed value stored in temporary variables generated by the compiler. The compiler determines the order of operations given by the three-address code.

Three-Address Code is Used in Compiler Applications:

- **Optimization:** Three-address code is often used as an intermediate representation during the optimization phases of compilation. It allows the compiler to analyze the code and perform optimizations that can improve the performance of the generated code.
- **Code Generation:** During the code generation phase, three-address code serves as an intermediate representation. This enables the

compiler to generate code that is specific to the target platform while ensuring that the generated code is correct and efficient.

- **Debugging:** Three-address code can be helpful in debugging the code generated by the compiler. Since it is a low-level language, it is often easier to read and understand than the final machine code. Developers can use three-address code to trace the execution of the program and identify errors or issues.
- **Language Translation:** Three-address code can also facilitate translating code from one programming language to another. By translating code to a common intermediate representation, it becomes easier to convert the code to multiple target languages.

General Representation

```
a = b op c
```

Where a, b or c represents operands like names, constants or compiler generated temporaries and op represents the operator

Example-1: Convert the expression a * – (b + c) into three address code.

$t_1 = b + c$
$t_2 = uminus\ t_1$
$t_3 = a * t_2$

Example-2: Write three address code for following code

```
for(i = 1; i<=10; i++)
 {
  a[i] = x * 5;
 }
```

$i = 1$
$L : t_1 = x * 5$
$t_2 = \&a$
$t_3 = sizeof(int)$
$t_4 = t_3 * i$
$t_5 = t_2 + t_4$
$*t_5 = t_1$
$i = i + 1$
$if\ i<=10\ goto\ L$

5.3 QUADRUPLE AND TRIPLE

There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

1. **Quadruple –** It is a structure which consists of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

   Advantage –

   - Easy to rearrange code for global optimization.
   - One can quickly access value of temporary variables using symbol table.

   Disadvantage –

   - Contain lot of temporaries.
   - Temporary variable creation increases time and space complexity.

Example – Consider expression a = b * – c + b * – c. The three address code is:

```
t1 = uminus c   (Unary minus operation on c)
t2 = b * t1
t3 = uminus c (Another unary minus operation on c)
t4 = b * t3
t5 = t2 + t4
a = t5  (Assignment of t5 to a)
```

#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

2. **Triples –** This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

   Disadvantage –

   - Temporaries are implicit and difficult to rearrange code.
   - It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

Example – Consider expression a = b * – c + b * – c

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

3. **Indirect Triples –** This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

   Example – Consider expression a = b * – c + b * – c

List of pointers to table

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Indirect Triples representation

Question – Write quadruple, triples and indirect triples for following expression : (x + y) * (y + z) + (x + y + z)

Explanation – The three address code is:

```
(1) t1 = x + y
(2) t2 = y + z
(3) t3 = t1 * t2
(4) t4 = t1 + z
(5) t5 = t3 + t4
```

#	Op	Arg1	Arg2	Result
(1)	+	x	y	t1
(2)	+	y	z	t2
(3)	*	t1	t2	t3
(4)	+	t1	z	t4
(5)	+	t3	t4	t5

Quadruple representation

#	Op	Arg1	Arg2
(1)	+	x	y
(2)	+	y	z
(3)	*	(1)	(2)
(4)	+	(1)	z
(5)	+	(3)	(4)

Triples representation

List of pointers to table

#	Op	Arg1	Arg2
(14)	+	x	y
(15)	+	y	z
(16)	*	(14)	(15)
(17)	+	(14)	z
(18)	+	(16)	(17)

#	Statement
(1)	(14)
(2)	(15)
(3)	(16)
(4)	(17)
(5)	(18)

Indirect Triples representation

5.4 SYNTAX DIRECTED TRANSLATION IN COMPILER DESIGN

The parser uses a Context-Free Grammar (CFG) to validate the input string and produce output for the next phase of the compiler. The output can be either a parse tree or an abstract syntax tree. To interleave semantic analysis with the syntax analysis phase of the compiler, we use Syntax Directed Translation (SDT).

Conceptually, with both syntax-directed definitions and translation schemes, we parse the input token stream, build the parse tree, and then traverse the tree as needed to evaluate the semantic rules at the parse tree nodes. Evaluating the semantic rules may generate code, save information in a symbol table, issue error messages, or perform other activities. The translation of the token stream is the result obtained by evaluating these semantic rules.

Definition

Syntax Directed Translation has augmented rules to the grammar that facilitate semantic analysis. SDT involves passing information bottom-up and/or top-down to the parse tree in form of attributes attached to the nodes. Syntax-directed translation rules use 1) lexical values of nodes, 2) constants & 3) attributes associated with the non-terminals in their definitions.

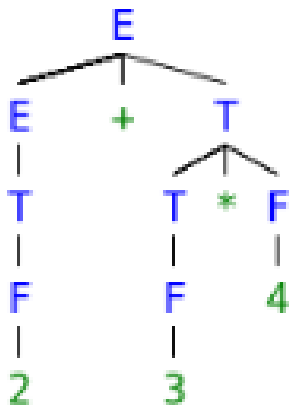
The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree.

Example

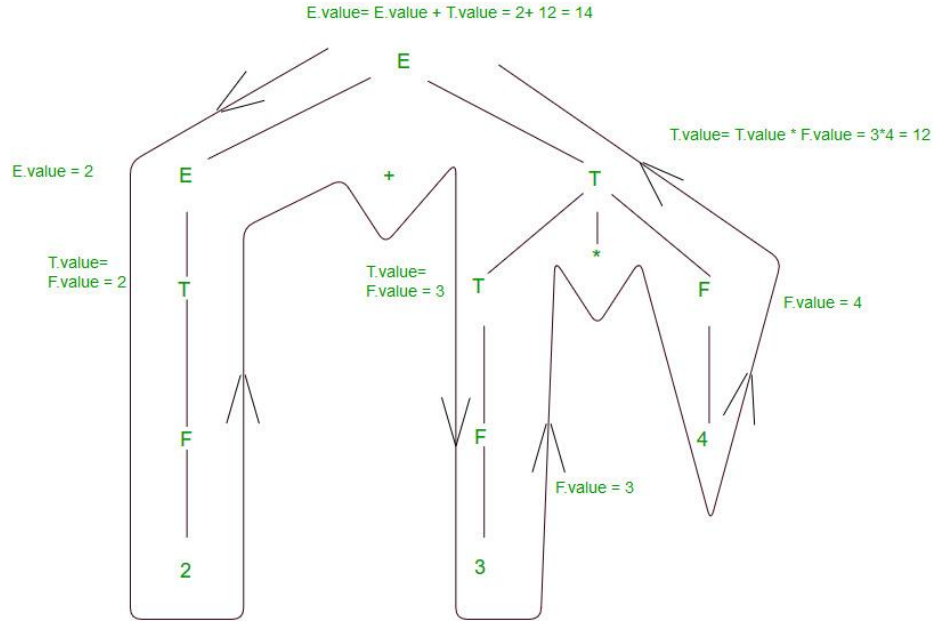
```
E -> E+T | T
T -> T*F | F
F -> INTLIT
```

```
E -> E+T     { E.val = E.val + T.val }   PR#1
E -> T       { E.val = T.val }           PR#2
T -> T*F     { T.val = T.val * F.val }   PR#3
T -> F       { T.val = F.val }           PR#4
F -> INTLIT  { F.val = INTLIT.lexval }   PR#5
```

Let's take a string to see how semantic analysis happens – S = 2+3*4. Parse tree corresponding to S would be



To evaluate translation rules, we can employ one depth-first search traversal on the parse tree. This is possible only because SDT rules don't impose any specific order on evaluation until children's attributes are computed before parents for a grammar having all synthesized attributes. Otherwise, we would have to figure out the best-suited plan to traverse through the parse tree and evaluate all the attributes in one or more traversals. For better understanding, we will move bottom-up in the left to right fashion for computing the translation rules of our example.



The above diagram shows how semantic analysis could happen. The flow of information happens bottom-up and all the children's attributes are computed before parents, as discussed above. Right-hand side nodes are sometimes annotated with subscript 1 to distinguish between children and parents.

5.5 SUMMARY

This chapter mainly focuses on Representation of Strings using Three Address Code, Syntax Directed Translation and Semantic Analysis

5.6 EXERCISE

Q. 1 Write a short note on Three Address Code.

Q. 2 What do you mean by Semantic Analysis?

Q. 3 Define Quadruple.

Q. 4 What are advantages of Syntax Directed Translation ?

5.7 REFERENCES

- Compilers: Principles, Techniques, and Tools" by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman 2nd Edition, Pearson Publication, 2006 ISBN-13: 978- 0321486813
- Modern Compiler Implementation in C" by Andrew W. Appel, 3rd Edition, Cambridge University Press, 2020, ISBN-13: 978-1108426631

- Principles of Compiler Design" by D. M. Dhamdhere, 2nd Edition Publisher: McGraw-Hill Education, 2017, ISBN-13: 978-9339204608
- https://www.geeksforgeeks.org/syntax-directed-translation-in-compiler-design/
- https://www.geeksforgeeks.org/three-address-code-compiler/

❄❄❄❄❄❄❄

6

CODE OPTIMIZATION

Unit Structure

6.0 Objectives

6.1 Introduction

6.2 Overview of Optimization
- 6.2.1 Code Optimization
- 6.2.2 Machine Dependent Optimization
- 6.2.3 Machine Independent Optimization

6.3 Loop Optimization Techniques

6.4 Data Flow Analysis

6.5 Code Generation Techniques
- 6.5.1 Target Machine description
- 6.5.2 Overview of Assembly process
- 6.5.3 Register Allocation
- 6.5.4 Instruction Selection

6.6 Summary

6.7 Exercise

6.8 References

6.0 OBJECTIVES

- To Examine evaluation time required by compiler to execute program.
- Outline program Readability and Maintainability
- To apply optimization techniques for code improvement.
- To construct the basic architecture of machine model.
- Formulate the need of register and memory allocation needed for compiler design.

6.1 INTRODUCTION

When we are writing program it is nothing but some kind of specifications which we are informing to compiler to do computation. After reading these specifications compiler will generate object code which will have some other specifications. Depending on the input there are many object programs which will follow some specifications. Some of the specifications may take lot of time to execute or they may take more memory. Therefore, there is a need to optimize code so that one can save memory or execution time.

6.2 OVERVIEW OF OPTIMIZATION

Code optimization which is also known as code improvement in compiler design is a critical phase that focuses on improving the performance and efficiency of the generated machine code without altering its functionality. This phase sits strategically after the initial parsing and semantic analysis stages, where the high-level code is converted into an intermediate representation (IR), and before the final code generation phase, where the IR is translated into machine code specific to the target architecture. The goal of code optimization is to produce a faster, more efficient program that consumes fewer resources, such as CPU time, memory, power etc.

This phase is optional and it is completely depending on compiler whether to execute program via this phase or omit this phase. Here code is transformed into some other form which may be easy to evaluate or process. Code optimization encompasses a broad spectrum of techniques and transformation, aimed at enhancing the performance and efficiency of code without altering its intended functionality. This process, crucial in compiler design and software development, involves multiple aspects that collectively contribute to the generation of optimized code. There are three criteria which can be considered as optimizing transformations:

1. Does the optimization capture most of potential improvement?
2. Does the optimization maintain the original meaning of program?
3. Does optimization reduce time or space of program?

To create an efficient target program a programmer needs more than an optimizing compiler which can take care of all above aspects. Following are the options available to programmer and compiler designer for creating efficient target program.

1. Criteria for code improving transformation

The transformation must preserve the meaning of program. It should not change the output by a program. Sometimes after optimization the program may slow down slightly so it should maintain its average speed.

2. Getting better performance

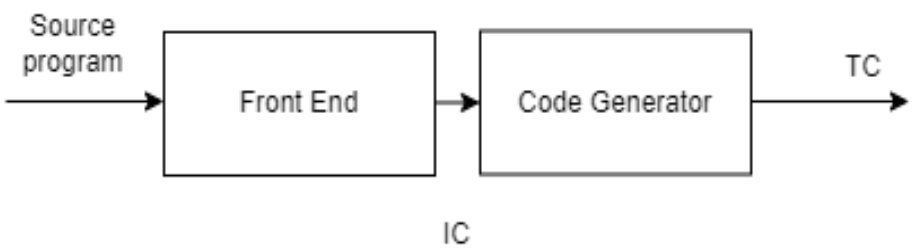


Figure 1 : representation of compiler

Here in this figure the Source program is submitted to front end of compiler where user can have freedom of using different algorithms, use of different loops like while, do-while , for and check the complexity of code.

While generating IC compiler can work on improving loops and memory address associated with variables. Compiler can be responsible to make good use of machine resources. Example. Keeping the most heavily used variable in register can cut down running time of program. In case of C program there is a provision of using storage class so that one can treat a memory location as register to speed up the execution process.

3. The organization of an optimizing compiler

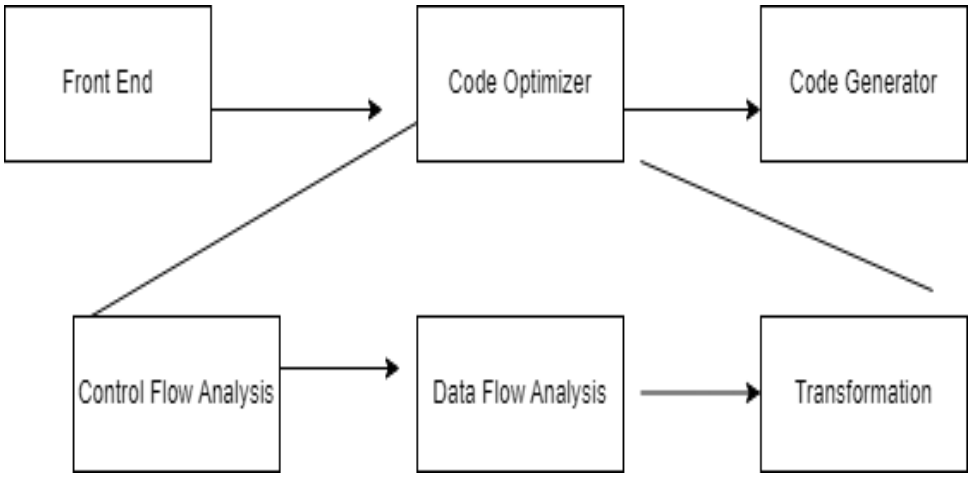


Figure 2: organizing optimizing compiler

Organizing an optimizing compiler involves structuring its components and workflow in a manner that efficiently processes source code to produce optimized machine code. An optimizing compiler typically goes through several stages, each responsible for different aspects of the translation and optimization process. The organization of these components is crucial for achieving effective optimization while maintaining the correctness of the compiled program. Various techniques are needed to transform program into a better version.

6.2.1 Code Optimization

Code optimization in the context of compiler design refers to the phase where the compiler attempts to improve the intermediate or final code it generates, making it run faster, consume less memory, or otherwise use system resources more efficiently without changing the semantics of the program. This optimization can occur at various stages of the compilation process and can target different aspects of the program's performance.

Impact of Code Optimization

Execution Speed: Perhaps the most sought-after result of code optimization is faster program execution. This can be achieved by eliminating unnecessary operations, improving the use of CPU caches, or reducing the overhead of function calls, among other techniques.

Resource Usage: Efficiently using memory and other system resources can not only speed up a program but also reduce its operational costs, especially in large-scale or embedded systems.

Energy Consumption: For mobile devices and data centers, optimized code can lead to significant savings in energy consumption, which is both economically beneficial and environmentally friendly.

User Experience: For end-users, optimizations can lead to more responsive applications and longer battery life on mobile devices, directly impacting the perceived quality of the software.

6.2.2 Machine Dependent Optimization

Machine-dependent as a name indicates that optimization is completely depend on machine model or architecture of machine and its components used like register, addressing modes etc. i.e. optimization in compiler design refers to the phase or set of transformations that specifically target the characteristics and features of the underlying hardware platform for which the code is being compiled. Unlike machine-independent optimizations that focus on language-level or algorithmic improvements applicable across different platforms, machine-dependent optimizations take into account the specifics of the target architecture to enhance performance. These optimizations can significantly impact the efficiency and speed of the compiled program by leveraging the unique capabilities and avoiding the specific limitations of the hardware.

Following are the examples of machine dependent optimization techniques:

a) Register Allocation

One of the primary machine-dependent optimizations is register allocation. Registers are the fastest storage available to a CPU, and efficient use of registers can significantly speed up a program. The compiler decides which variables or intermediate values should be kept in these limited but fast storage locations. Advanced register allocation algorithms, like graph coloring, are used to make these decisions effectively.

Example:

Consider the program fragment in high level language

```
int a = 5;

int b = 10;

int c = a + b;

int d = c * 2;
```

Without optimization	After optimization
MOV [a], 5 ; Move 5 into memory location 'a' MOV [b], 10 ; Move 10 into memory location 'b' MOV R1, [a] ; Load 'a' from memory into register R1 MOV R2, [b]; ADD R1, R2 MOV [c], R1 ; Store result from R1 to 'c' MOV R3, [c]; Load 'c' from memory R3 SHL R3, 1 ; shift left by 1 bit MOV [d], R3; Store result from R3 to memory location 'd'	MOV R1, 5 ; Move 5 directly into register R1 (for 'a') MOV R2, 10 ; Move 10 directly into register R2 (for 'b') ADD R1, R2 ; Add R1 and R2, result in R1 ('c' = 'a' + 'b') SHL R1, 1 ; Multiply R1 by 2 (shift left by 1 bit, for 'd')

Without register allocation optimization, the compiler might naively store all variables in memory and load them into registers only when an operation is performed. Without register allocation optimization, the compiler might naively store all variables in memory and load them into registers only when an operation is performed.

Advantages :

- Fast accessible storage
- Allows computations to be performed on them
- Deterministic behavior
- Reduce memory traffic
- Reduces overall computation time

Disadvantages :

- Registers are generally available in small amount (BC DE HL in case of 8085 micro processor)
- Register sizes are fixed and it varies from one processor to another
- Registers are complicated
- Need to save and restore changes during context switch and procedure calls

b) Instruction Selection

Different CPUs support different sets of instructions, with some instructions being more efficient than others for certain tasks. Instruction selection optimization involves choosing the most efficient machine instructions to perform operations represented in the intermediate code. This might include using specialized instructions for certain mathematical operations, memory access patterns, or data manipulation tasks that are unique to the processor architecture.

Instruction selection is a crucial machine-dependent optimization process in compilers, where the compiler chooses the most efficient machine instructions to implement high-level language constructs. This optimization ensures that the generated machine code makes optimal use of the target architecture's instruction set and features. Example :

Consider program fragment

int a = 1;

int b = 1;

int c = a + b;

Before optimization	**After optimization**
MOV eax, 5 ; Move 5 into register eax (a) MOV ebx, 10 ; Move 10 into register ebx (b) ADD eax, ebx ; Add a and b, result in eax (c)	MOV eax, 5 ; Move 5 into register eax (a) ADD eax, 10 ; Add 10 directly to eax (b), result in eax (c)

With instruction selection optimization, the compiler can leverage the specific features of the x86 architecture to produce more efficient code.

c) Instruction Scheduling

The order in which instructions are executed can greatly affect performance, especially on modern CPUs with complex pipelines and execution units capable of parallel instruction execution. Instruction scheduling rearranges the order of instructions to avoid pipeline stalls (waiting states) and to make efficient use of instruction-level parallelism. This optimization must consider the CPU's specific pipeline architecture and execution dependencies.

The pipelined architecture allows multiple instructions to be executed simultaneously, with different stages of each instruction executed concurrently in different pipeline stages.

Instruction scheduling aims to reorder instructions to minimize dependencies and stalls, ensuring that the pipeline operates at maximum throughput.

The compiler may analyze the dependency between instructions and reorder them to minimize pipeline stalls caused by data hazards or resource conflicts.

In the optimized version, the instructions are already in a sequence where the output of one instruction is not needed immediately by the next, minimizing stalls and maximizing the pipeline's utilization.

6.2.3 Machine Independent Optimization

Machine-independent optimization in compiler design refers to a set of optimizations that can be applied to source code regardless of the target machine architecture. These optimizations focus on improving the efficiency and performance of programs at a high-level language representation, such as intermediate code or abstract syntax trees, without considering specific hardware details.

Example

a) Elimination of common sub expression

Common Subexpression Elimination (CSE) is a machine-independent optimization technique that identifies redundant computations within a program and eliminates them to improve performance. It involves identifying expressions that are computed multiple times within a program and replacing them with a single computation, storing the result in a temporary variable.

Consider the statement *cost=2*rate+(start-finish-100)+(start-finish+rate)*

Three address code for the above statement is

*T1=2*rate*

T2=start-finish

T3=T2-100

T4=start-finish

T5=T4+rate

T6=T1+T4

T7=T6+T5

Cost=T7

Here start-finish is repeated so we can eliminate one of the statement and can optimize the code as follows

*T1=2*rate*

T2=start-finish

T3=T2-100

T4=T3+T1

T5=T2+rate

cost=T5+T4

b) Constant folding

Constant folding is a machine-independent optimization technique where the compiler evaluates constant expressions at compile time instead of deferring their evaluation until runtime. It involves replacing expressions composed entirely of constants with their computed values.

Example:

Consider statement *int a = 10 + 5;*

In this code, the expression 10 + 5 is a constant expression because both operands are literals. During compilation, constant folding can be applied to evaluate the expression 10 + 5 and replace it with its computed value.

After constant folding, the code becomes:

int a = 15;

c) Dead code elimination

A piece of code is said to be dead if the results evaluating the code are not used in the program , such code can be eliminated safely. It helps in reducing the size of the compiled program and improving runtime efficiency by eliminating unnecessary computations and memory allocations.

Example:

A=25 *{* *Lines of code* *A=b+c* *Lines* *}*	Here , A=25 is dead since its value is updated so we can improve the code as follows *{* *Lines of code* *A=b+c* *Lines* *}* *A=25*

{ *int x = 5;* *int y = 10;* *if (x < y) {* *printf("x is less than y\n");* *} else {* *printf("x is greater than or equal to y\n");* *}* *}*	Here x and y are initialized so always x is less than y will be executed and else part will never going to execute. We can eliminate the same and code will become *{* *int x = 5;* *int y = 10;* *if (x < y) {* *printf("x is less than y\n");* *}* *}*

d) Usage of high operators over low operators

Benefits of Using Addition Instead of Multiplication:

Efficiency: Addition operations are generally faster than multiplication operations, especially on processors with limited hardware resources.

Simplicity: The code becomes more concise and easier to understand by replacing multiple addition operations with a single multiplication.

Reduced Overhead: The compiler may optimize the multiplication operation further, depending on the target architecture, resulting in reduced overhead.

Similarly, Division operations are generally faster than subtraction operations on modern processors. Processors are optimized to perform division efficiently, especially for division by constant values.

6.3 LOOP OPTIMIZATION TECHNIQUES

Before we understand loop optimization techniques let us understand what is loop in programming language. Loops are nothing but one form of control structure which allows block of statements to be executed until certain condition is fulfilled. Loop consists of path from top to bottom. Here top of loop is known as header(H) and path (P) specifies the route which one needs to follow till certain conditions are fulfilled denoting as loop(H, P).

Optimizing loops is particularly important in compilation, since loops (inner loops) account for much of the executions times of many programs. Since tail-recursive functions are usually also turned into loops, the importance of loop optimizations is further magnified.

Loop is very important place when optimization is necessary, the inner loops where program tend to spend more time. The running time of program may be improved if we decrease the number of instructions in an inner loop. Loop Optimization is the process of increasing execution speed and reducing the overheads associated with loops. It plays an important role in improving cache performance and making effective use of parallel processing capabilities. Most execution time of a scientific program is spent on loops.

1) Code Motion (Frequency Reduction)

Here, the amount of code in the loop is decreased. A statement or expression, which can be moved outside the loop body without affecting the semantics of the program, is moved outside the loop.

Example:

Before optimization	**After optimization**
while(i<100) *{* *a = Sin(x)/Cos(x) + i;* *i++;* *}*	t = Sin(x)/Cos(x); while(i<100) { a = t + i; i++; }

Here in this example always value of sin(x) and cox(x) will be same so instead of keeping statement inside loop we can move it outside the loop i.e. beginning loop to reduce the time required to compute.

2) Induction Variable Elimination

If the value of any variable in any loop gets changed every time, then such a variable is known as an induction variable. With each iteration, its value either gets incremented or decremented by some constant value.

3) Loop Unrolling

Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.

Before optimization	**After optimization**
I=1 While(I<=100)	I=1 While(I<=100)

{ X[i]=0; I++; }	{ X[i]=0; I++; X[i]=0; I++ }

Here, i<=100 will be performed 100 times but if the body of loop is replaced then number of times this test is performed could be reduced. Unrolling makes 2 copies of body so that work can be reduced to 50%.

4) Loop jamming

Loop jamming is combining two or more loops in a single loop. It reduces the time taken to compile the many loops.

Before optimization	**After optimization**
for(int i=0; i<5; i++) a = i + 5; for(int i=0; i<5; i++) b = i + 10;	for(int i=0; i<5; i++) { a = i + 5; b = i + 10; }

Here, we merge the bodies of loop.

6.4 DATA FLOW ANALYSIS TECHNIQUES

As a name indicates this technique involves the flow of data in control flow graph, i.e. the study helps us to determine the information regarding the definition and for what purpose data is used in the program. This method helps in optimization as flow of the data helps to understand it's movement. One can trace the value or variable and can find out how the variable is changing its value based on instructions written. It is very similar to add a watch on variables in 'C' Program and with the help of F7 key one can find or trace the variable so that flow and hence logical error can be traced.

In order to implement technique, we can design graph in the form of flowchart representing node as program statements and edges as flow between statements. One can use rules and regulations to compute values of each expression and variables associated with them.

Following is a list of some of the common types of data flow analysis

1. <u>Reaching Definitions Analysis</u>:

As the name indicated reaching definition implies whether a variable or expression can be reached with the help of some logical programming. If a particular variable is unable to reach it implies we can remove that variable as it is never going to be the part of program.

Example

A definition D is reaches a point x if there is path from D to x in which D is not killed, i.e., not redefined.

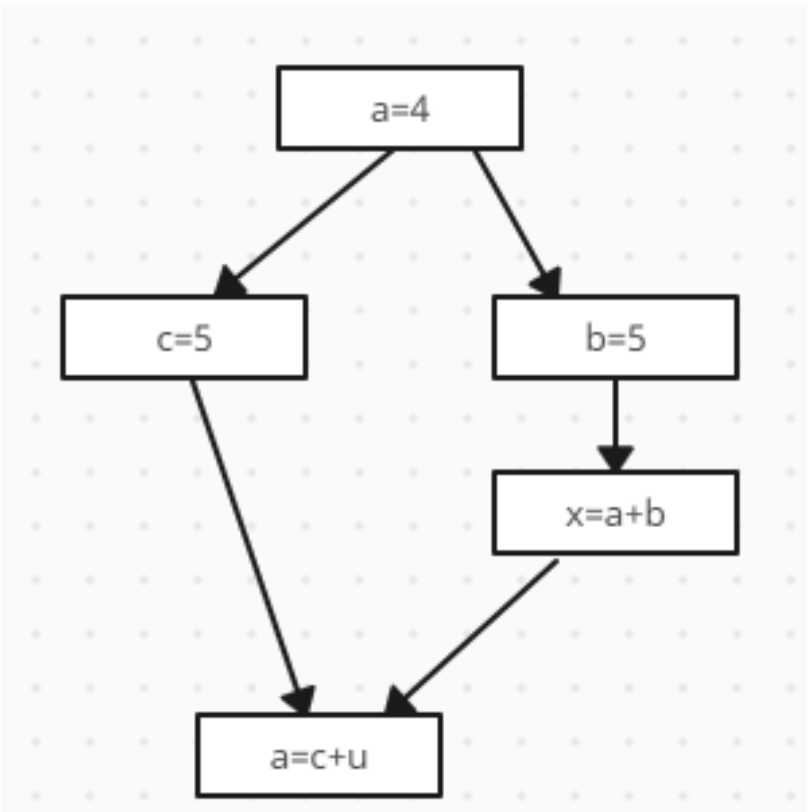
X=0 *this is block 1 where X is initialized to 0*

X=X+7 *This is block 2 where value of X is updated*

Y=X+7 *this is block 3 here value from block 1 is not accessible.*

2. <u>Live Variable Analysis</u>: This analysis find the points in program where variable is holding some value which may come from some computing operations or it is taking part in some computation. If it is not taking any part of data movement then one can safely eliminate.

Example : A variable is said to be live at some point p if from p to end the variable is used before it is redefined else it becomes dead.



3. Available Expressions Analysis:

This analysis can be used to find whether a particular expression is taking part in evaluating expression which helps in eliminating common sub expression.

4. Constant Propagation Analysis:

Constants play a vital role in programming and hence to keep a track of such constants and optimize the work we can use this techniques. We can track values of constants and find point in the code where they are used.

6.5 CODE GENERATION TECHNIQUES

Code generation techniques can be the final stage of compiler activity. The code generation of high level language is nothing but the object code of that language. For example in case of JAVA we get .class files based on number of classes present in the program. The .class files are nothing but object files. Code generation process is very tricky due to its complex operations as compiler has to deal with various forms of instructions based on addressing modes. The architectural issues may be discussed with respect to registers and accumulators. Selecting proper instructions is also an important feature to optimize code.

They should have following properties:

a. It should preserve the meaning of original problem.

b. It should be efficient with respect to CPU and memory management.

6.5.1 Target Machine description

Target code generation is one of the important aspect in converting assembly level language to optimized code into machine understandable format. Target code can be machine readable code or assembly code or X86 instruction format. Here the machine will read each and every line and it will convert into its corresponding numerical opcode format and the conversion is always in 1:1 mapping. Like each instruction in X86 format will have corresponding one code in numerical code format.

While generating code on target machine one should look for following properties of machine in the form of its design or architecture as most of time instruction will be using registers as they are the fast in performing many operations. As registers are the internal part of CPU they are limited in number and size as well.

1) Instruction Set

Every X86 supporting languages will have their own design and hence they are machine dependent and hence their instructions may vary depending upon what kind of bits they use. Like 8 bit, 16 bit and so on. Variety of instruction types are available like arithmetic, logical, conditional, data or

block transfer etc. Some instructions are like increment and decrement which allows to increase the data of block by 1 or decrease data of block by 1.

Following table shows some of the instruction based on category.

Format of operations	Examples of instructions
Conditional	JZ (jump if zero) JC (jump if carry)
Arithmetic	ADD (adds two numbers) SUB (Subtracts two numbers)
Block transfer/ Data transfer	MOV (moves data from source to destination) LDA (load data)

Table 1 : examples of operations

Here when we perform any operation always data will come from accumulator and result will be stored in accumulator.

2) Addressing Modes

Addressing modes define in what way data will come to system and how it will get processed by the system. Following are the different modes of addressing.

Addressing mode	Examples
Register to register	MOV A, B Here A and B are registers as they are oprands.
Immediate	MVI A,05H Here number 5 will be transferred to register.
Direct	LDA A,1000H Here 1000 is a memory address. Content from memory address 1000 is extracted and stored in register A

Table 2: Sample of addressing modes

3) Instruction Formats

The format of instruction will talk about how one should write instruction while coding. General format is as follows:

[Label] Opcode [operand/s]

Here label can be optional and used only if there are conditional statements written.

Operands can be optional as one can use it for auto increment and auto decrement purpose.

Maximum 2 operands can be specified.

6.5.2 Overview of Assembly Process

The process of converting mnemonics into low level language is nothing but assembly process in which system performs following operations:

1) Scan instruction and create tokens based on opcode , operands etc.
2) Identify symbols/variables and enter them in symbol table.
3) Identify literals if any and put them into literal table.
4) Keep updating location counter.
5) Allocate memory to variables
6) Scan instruction and check whether it is there in opcode table. Check syntax by mapping character by character. If any error is found reject.
7) Perform semantic check on instruction.
8) Extract numerical opcode and extract memory address of variable defined.
9) Generate instruction.

Example:

Consider following code

LC	Instruction
. . . 13 . . . 25	. . . A DS 1 . . . L1: ADD N

Assume that there is a declaration statement on line number 13 stating that declare a block A. Later on line number 25 says that Add variable N.

Assembler will create tokens as

A	DS	1

As soon as it detects variable A it will be added in symbol table. When it fetches instruction from line 25 it will identify there is an instruction ADD. It will verify the same with the help of Opcode table and extract corresponding code and generate instruction.

6.6 SUMMARY

Code optimization is a critical process for enhancing the efficiency of software. It involves a careful trade-off between improving performance and maintaining other important attributes such as readability and maintainability. Effective optimization requires a deep understanding of both the software being written and the hardware on which it will run. It's also a cooperative process between the programmer and the compiler, each bringing its strengths to produce the most efficient code possible while preserving the program's semantics.

6.7 EXERCISE

1. Justify importance of optimization in compiler.
2. Demonstrate any two techniques of machine independent optimization.
3. Elaborate loop unrolling and loop jamming techniques.
4. Compare machine dependent and machine independent optimization techniques.

6.8 REFERENCES

- Compilers: Principles, Techniques, and Tools" by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman 2nd Edition, Pearson Publication, 2006 ISBN-13: 978- 0321486813
- Modern Compiler Implementation in C" by Andrew W. Appel, 3rd Edition, Cambridge University Press, 2020, ISBN-13: 978-1108426631
- Principles of Compiler Design" by D. M. Dhamdhere, 2nd Edition Publisher: McGraw-Hill Education, 2017, ISBN-13: 978-9339204608
- https://www.geeksforgeeks.org/syntax-directed-translation-in-compiler-design/

  https://www.geeksforgeeks.org/three-address-code-compiler/

❄❄❄❄❄❄❄

7

RUNTIME ENVIRONMENTS

Unit Structure

7.0 Objective

7.1 Introduction

7.2 Activation Records and Stack Management

- 7.2.1 Introduction to Activation Records
  - 7.2.1.1 Structure of an activation record
  - 7.2.1.2 Role in function/procedure call management
- 7.2.2 Stack Management
  - 7.2.2.1 Call stack and its significance
  - 7.2.2.2 Stack frame allocation and deallocation
  - 7.2.2.3 Stack pointer and frame pointer management

7.3 Heap Memory Management

- 7.3.1 Difference between stack and heap memory
- 7.3.2 Dynamic Memory Allocation
  - 7.3.2.1 Allocation and deallocation techniques
  - 7.3.2.2 Garbage collection methods (reference counting, mark-and-sweep, generational GC)

7.4 Call and Return Mechanisms

- 7.4.1 Call Mechanism
- 7.4.2 Return Mechanism

7.5 Exception Handling

7.6 Lexical and Syntax Error Handling

- 7.6.1 Lexical Error Handling
  - 7.6.1.1 Introduction to Lexical Errors
  - 7.6.1.2 Error Recovery Strategies
  - 7.6.1.3 Error Reporting and Handling
- 7.6.2 Syntax Error Handling
  - 7.6.2.1 Introduction to Syntax Errors
  - 7.6.2.2 Error Recovery Strategies
  - 7.6.2.3 Error Reporting and Handling

7.7 Summary

7.8 Questions for practice

7.9 References

7.0 OBJECTIVE

The objectives of this study material are to provide a foundational understanding of activation records, stack management, and heap memory management, including dynamic memory allocation and garbage collection techniques. Learners will master call and return mechanisms, such as parameter passing and function call conventions, and gain knowledge of exception handling, including try-catch blocks and stack unwinding. Additionally, the material aims to equip learners with the skills to identify, report, and recover from common lexical and syntax errors using appropriate strategies, enhancing their problem-solving abilities in compiler design.

7.1 INTRODUCTION

This study material on the principles of compiler design covers key concepts such as runtime environments, activation records, stack and heap memory management, call and return mechanisms, and exception handling. Additionally, it addresses lexical and syntax error handling, providing strategies for error recovery and reporting. The aim is to equip learners with a solid foundation in compiler design and practical skills for managing errors and optimizing performance.

7.2 ACTIVATION RECORDS AND STACK MANAGEMENT

Activation records and stack management are fundamental components in the execution of function and procedure calls in a program. An activation record, or stack frame, stores vital information for each active subroutine, including local variables, return addresses, parameters, and saved registers. These records are pushed onto the call stack when a function is called and popped off when the function returns, maintaining the correct state of the program's execution. Effective stack management involves the careful allocation and deallocation of stack frames, ensuring that memory is used efficiently and that the call stack accurately reflects the program's call history. This mechanism supports function call conventions, such as parameter passing and return value handling, and is crucial for enabling recursion and nested function calls. Understanding activation records and stack management is essential for optimizing program performance, ensuring efficient memory usage, and facilitating effective debugging and error handling.

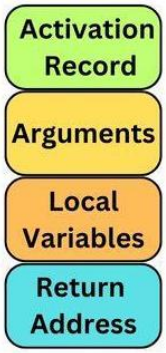
7.2.1 Introduction to Activation Records

Activation records, also known as stack frames, are critical data structures used by compilers to manage information needed during function or procedure calls in a program. Each activation record contains essential data such as the function's local variables, arguments passed to the function, the return address, and saved registers. When a function is called, an activation record is created and pushed onto the call stack, and when the function returns, this record is popped off the stack. This process ensures that each function's execution context is maintained correctly, allowing for proper

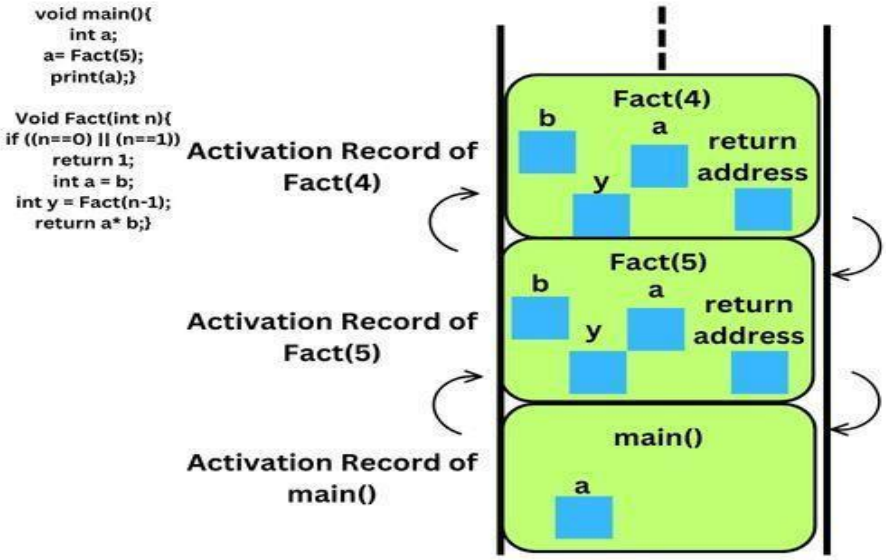
control flow and enabling features like recursion and nested calls. Understanding activation records is vital for grasping how function calls are handled, how memory is managed during execution, and how debuggers track the state of a program.

7.2.1.1 Structure of an activation record

When a procedure gets called, the computer creates an activation record to store all the information needed to execute that procedure. This information includes the procedure's arguments, local variables, and return address. When the operation finishes executing, the computer deletes the activation record. Activation record is also known as stack frames or function call frames used by the compiler to manage the execution of a function or procedure.



Imagine a scenario where you have a program with multiple functions. A new activation function gets created whenever one of the functions gets called. The activation record is stored on the control stack whenever a process gets executed. The control stack is a runtime stack used to track live procedure activations. Its primary purpose is to determine which execution still needs to be completed. As the activation begins, the procedure name is pushed into the stack and will pop out as the activation ends. If there is a recursive procedure, then several activations are active at the same time. If there is a non-recursive procedure, one activation of the function is executed simultaneously.



Example

Consider a simple example to understand the activation record concept better. Here in this code, we have taken a function named "addition" that returns the result of adding two numbers.

```
#include <stdio.h>

// function to perform addition

void addition(int a, int b) {

    int result = a + b;

    printf("Result: %d\n", result);

}

//main program

int main() {

    int x = 2;

    int y = 8;

    addition(x, y);

    return 0;

}
```

Output

```
Result: 10
```

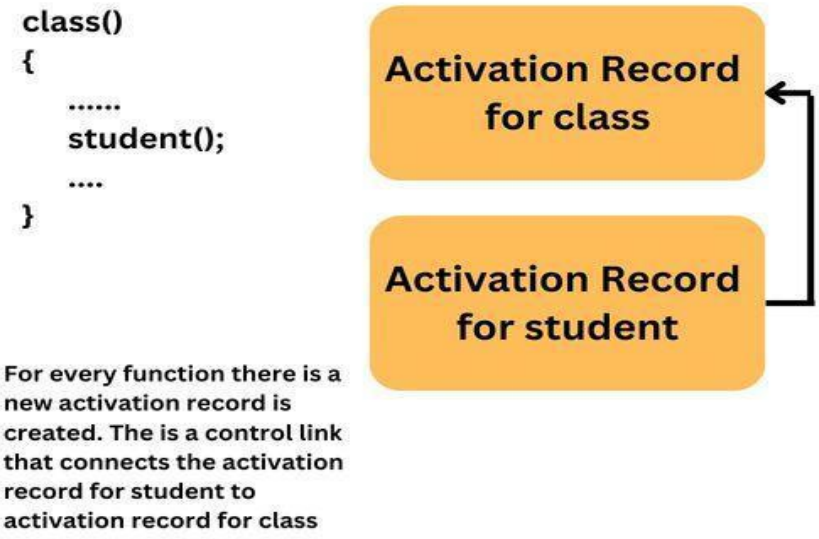
Explanation

The 'main' function is the program's starting point. It consists of local variables like 'x' and 'y'. When the 'addition' function gets called from the main function, a new activation record gets created for the 'addition' function. The activation record of 'addition' is initialized with parameters with 'a' and 'b,' which have (2 and 8) values from the calling function. The result gets stored with the value of 10. Once the 'addition' function work is completed and reaches its end, then activation records are removed from the calling stack. Finally, when the 'main' function terminates, its activation record is removed from the stack.

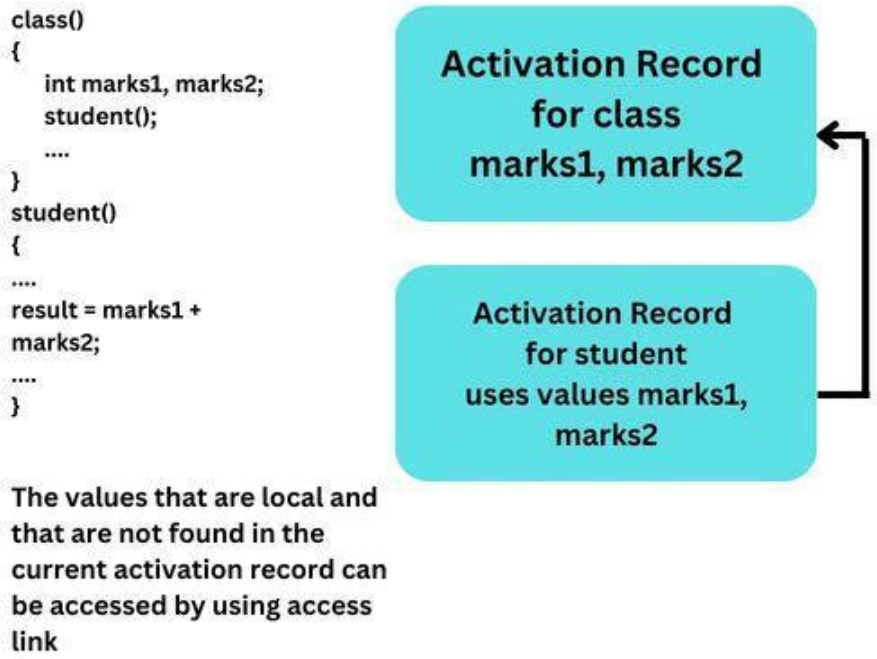
a. **Return Address:** This address holds the location where the control should return upon task completion. This feature helps the program continue executing from the same point it was initially created. It is used by a calling function that will return a value to the same calling function.

b. **Parameters:** The calling procedure uses it to supply parameters to the called method. It stores actual parameters used to send input to the called system. The parameters can be passed by value or reference and stored in the activation record for the function to access.

c. **Control Link:** It points toward the activation record of the caller. It allows you to return and execute continuously. The system uses it to store information outside the local scope. The control link connects the activation record to the activation record of the caller.



d. **Access Link :** It stores the address of the activation record of the caller function.



e. **Saved Machine Status:** The activation record consists of critical information about the program's state, which is just about to get called. It stores information like the return address or machine registers. The saved machine ensures the program can resume execution once the procedure call gets terminated.

f. **Local Data:** This field consists of local data for a particular function's execution. Local data consists of variables that serve the purpose of quick calculations or storing specific values of a currently used function.

g. **Temporaries:** It refers to the variables or storage locations used to store intermediate values within the procedure. When a function executes, it may perform different operations that require temporary storage. Once the procedure call completes and the control returns to the calling code, the system deallocates activation records and releases temporaries.

7.2.1.2 Role in function/procedure call management

In compiler design, the role in function/procedure call management is critical for translating high-level programming languages into machine code or intermediate representations. Here are key roles related to function/procedure call management in compiler design:

a. **Parsing and Syntax Analysis:** This role involves parsing the source code to identify function and procedure calls, along with their

parameters and arguments. Syntax analysis ensures that the calls follow the grammar and syntactic rules of the programming language.

b. **Symbol Table Management:** Managing a symbol table is crucial for function/procedure call management. The symbol table keeps track of all declared functions, procedures, variables, and their associated information (e.g., data types, scope). During function/procedure calls, the compiler uses the symbol table to resolve identifiers and check for semantic correctness.

c. **Type Checking:** Ensuring type compatibility during function/procedure calls is another important role. The compiler checks that the types of arguments passed to functions/procedures match the expected parameter types, helping to catch type-related errors early in the compilation process.

d. **Intermediate Code Generation:** After parsing and semantic analysis, compilers often generate intermediate code representations. Function and procedure calls in the source code are translated into intermediate code instructions, which may involve managing activation records (stack frames) for each function/procedure call to handle parameters, local variables, and return addresses.

e. **Optimization:** Function/procedure call management plays a role in optimization strategies. Compilers may optimize function calls by inlining small functions, eliminating redundant calls, or optimizing parameter passing mechanisms (e.g., using registers for passing arguments efficiently).

f. **Code Generation:** Finally, during code generation, the compiler translates the intermediate code or abstract syntax tree into target machine code or assembly language. Function and procedure calls are translated into appropriate machine instructions, taking into account calling conventions, parameter passing mechanisms, and stack management.

g. Overall, function/procedure call management in compiler design encompasses parsing, semantic analysis, symbol table management, type checking, code generation, and optimization, all aimed at producing efficient and correct executable code from high-level programming languages.

7.2.2 Stack Management

In compiler design, stack management plays a crucial role in handling function calls, local variables, and control flow during program execution.

7.2.2.1 Call stack and its significance

In the context of compiler design, the call stack is a critical concept that impacts various aspects of program execution and memory management. Here's how the call stack is significant in compiler design:

a. **Function Calls and Control Flow:** When a compiler processes source code, it generates instructions for function calls and returns. These instructions manipulate the call stack to manage the flow of control during program execution.

The call stack ensures that function calls are handled in a structured manner, with each function call creating a new stack frame and returning control to the caller upon completion.

b. **Activation Records and Stack Frames:** Compiler design involves defining the structure of activation records (stack frames) for functions and procedures. This includes specifying the layout of parameters, local variables, return addresses, and other relevant information within each activation record.

The compiler generates code to manage stack frames, such as allocating space for local variables, passing parameters, and saving/restoring registers as needed.

c. **Parameter Passing Mechanisms:** The call stack plays a role in parameter passing mechanisms defined by the compiler. This includes strategies like passing parameters via registers, the stack, or a combination of both, depending on the calling conventions and architecture targeted by the compiler.

Stack-based parameter passing involves pushing parameters onto the stack before a function call and accessing them within the function through the corresponding stack offsets.

d. **Recursion Handling:** Compilers must handle recursion efficiently using the call stack. Recursive function calls create nested stack frames, allowing recursive algorithms to work correctly without causing stack overflow errors.

The compiler ensures that recursive calls properly manage stack space and stack frame layout to prevent excessive memory usage and maintain program integrity.

e. **Exception Handling and Error Reporting:** Compiler-generated code for exception handling often relies on the call stack. When an exception occurs, the call stack provides information about the function call hierarchy, helping to unwind the stack and locate appropriate exception handlers.

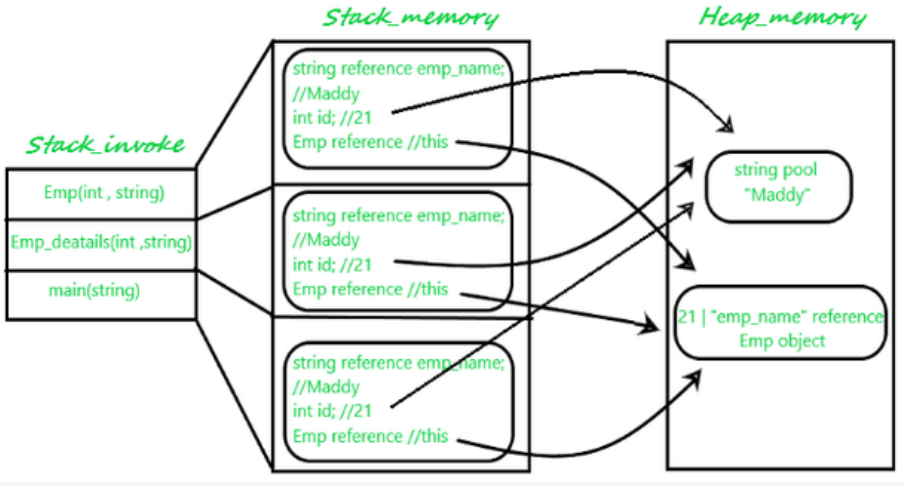
Error reporting mechanisms, such as stack traces, use the call stack to identify the sequence of function calls leading to an error, aiding developers in debugging and diagnosing issues in their code.

f. **Optimization Opportunities:** Advanced compilers employ stack-related optimizations to improve program performance. This includes techniques like stack frame reuse, stack slot allocation optimization, and tail call optimization to minimize stack overhead and improve execution efficiency.

7.2.2.2 Stack frame allocation and deallocation

In compiler design, stack allocation is a runtime storage management technique that uses a last-in, first-out (LIFO) principle for allocations and deallocations. The compiler calculates how much memory to allocate for each variable in the program, and the memory is automatically released when a function call is complete. This automatic deallocation is called stack unwinding, and it involves adjusting the stack pointer, which is a constant-time operation

In stack allocation, a contiguous area of memory is reserved for the stack, and pointers called the stack base (SB) and top of stack (TOS) point to the first and last entries, respectively. Activation records are pushed and popped onto the stack as activations begin and end, and each activation record contains storage for the locals in that call.



7.2.2.3 Stack pointer and frame pointer management

In compiler design and low-level programming, managing the stack pointer (SP) and frame pointer (FP) is crucial for efficient memory allocation and function call management. Here's how stack pointer and frame pointer management works and why it's significant:

a. **Stack Pointer (SP):** The stack pointer is a register or memory location that points to the top of the stack, indicating the current position for stack operations.

When a program starts or a function is called, the stack pointer is typically initialized to the top of the stack memory region.

Stack operations such as pushing data onto the stack (e.g., function parameters, return addresses) or popping data off the stack are performed by adjusting the stack pointer accordingly.

b. **Frame Pointer (FP):** The frame pointer is another register or memory location used specifically for accessing variables and data within the current stack frame (activation record).

Unlike the stack pointer, which moves dynamically during stack operations, the frame pointer remains fixed within a stack frame once it's set.

The frame pointer is particularly useful for accessing local variables and parameters within a function, as it provides a stable reference point within the stack frame.

c. **Function Call and Stack Frame Setup:** When a function is called, the compiler generates code to set up the stack frame (activation record) for that function.

d. **This setup process involves adjusting both the stack pointer (SP) and frame pointer (FP):** The stack pointer is moved downward to allocate space for function parameters, local variables, return address, and other control information.

The frame pointer is set to the base of the current stack frame, providing a stable reference for accessing variables within the frame.

e. **Stack Frame Usage:** Within a function, the frame pointer is used to access parameters, local variables, and other data stored in the current stack frame.

Accessing variables via the frame pointer avoids the need to adjust the stack pointer dynamically for each variable access, which can be more efficient in terms of code generation and execution speed.

f. **Stack Unwinding and Return:** When a function completes its execution or returns, the compiler generates code to unwind the stack frame and restore the previous execution context.

This process involves popping the current stack frame off the stack by adjusting the stack pointer and possibly restoring the previous frame pointer if necessary.

g. **Optimizations and Efficiency:** Efficient management of the stack pointer and frame pointer is critical for optimizing code size and execution speed.

Compilers may apply optimizations such as frame pointer omission (FPO) or using a combination of frame pointer and stack pointer for efficient variable access and function call management.

h. **Debugging and Stack Traces:** Stack pointer and frame pointer management are essential for debugging tools and stack traces that provide insights into program execution and function call hierarchies.

Tools like debuggers use the stack pointer and frame pointer information to display stack frames, local variables, and function call paths during program debugging.

7.3 HEAP MEMORY MANAGEMENT

Heap memory management in compiler design refers to how dynamically allocated memory is handled during program execution. Unlike stack memory, which is used for function calls and local variables, heap memory is used for dynamic data structures such as arrays, linked lists, objects, and other data that needs to be allocated and deallocated at runtime.

Here are key points about heap memory management in the context of compiler design:

a. **Dynamic Memory Allocation:** Heap memory allows programs to allocate memory dynamically at runtime, unlike stack memory where the size is typically fixed or determined at compile time.

Languages like C, C++, and others use functions like malloc, calloc, realloc, and free for heap memory management.

b. **Heap Data Structures:** Data structures such as arrays, linked lists, trees, hash tables, and objects are often allocated on the heap.

These data structures can grow and shrink dynamically based on program needs, making heap memory essential for managing complex data.

c. **Memory Allocation Algorithms:** Heap memory management involves algorithms for efficient allocation and deallocation of memory blocks.

Common algorithms include first-fit, best-fit, worst-fit, and buddy allocation, each with its trade-offs in terms of memory fragmentation, overhead, and allocation speed.

d. **Heap Fragmentation:** Fragmentation can occur in heap memory when allocated memory blocks become scattered, leading to inefficient use of memory.

Compilers and memory management libraries often employ strategies like memory compaction, defragmentation, and memory pooling to mitigate fragmentation issues.

e. **Memory Leaks:** Heap memory management includes handling memory leaks, which occur when allocated memory is not properly deallocated after use.

Memory leaks can lead to a gradual increase in memory consumption over time, potentially causing performance issues and resource exhaustion.

f. **Garbage Collection (GC):** Some programming languages, such as Java, C#, and Python, use garbage collection to automatically manage heap memory.

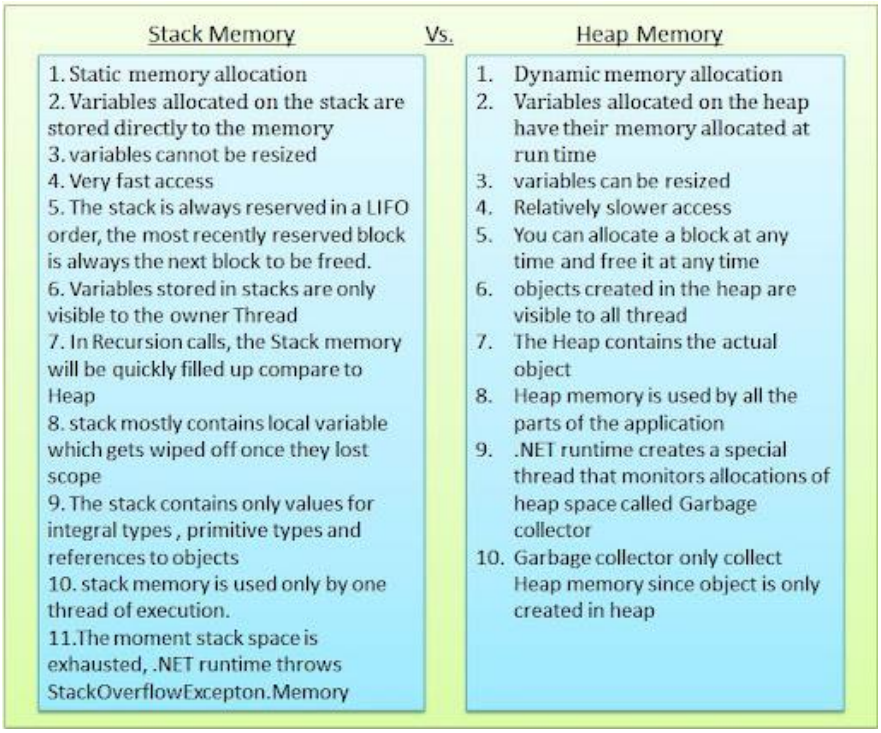
Garbage collection algorithms identify and reclaim unused memory (garbage) to free up heap space for future allocations.

g. **Manual Memory Management:** In languages like C and C++, developers must manually manage heap memory by allocating and deallocating memory using functions like malloc and free.

Manual memory management requires careful handling to avoid memory leaks, dangling pointers, and other memory-related errors.

h. **Compiler Optimizations:** Compilers may optimize heap memory usage by analyzing memory allocation patterns and applying optimizations such as object pooling, stack allocation for temporary objects, and optimizing memory access patterns.

7.3.1 Difference between stack and heap memory



7.3.2 Dynamic Memory Allocation

Dynamic memory allocation refers to the process of allocating memory for data structures or variables at runtime, as opposed to static memory allocation where memory is allocated at compile time. In the context of compiler design and programming languages, dynamic memory allocation is a fundamental concept that allows programs to manage memory flexibly based on runtime requirements.

7.3.2.1 Allocation and deallocation techniques

A. Allocation Techniques:

a. **Static Allocation:** Memory is allocated at compile time and remains fixed throughout the program's execution.

Typically used for global variables, constants, and static arrays.

Example: int staticArray[100];

b. Dynamic Allocation (Heap Allocation): Memory is allocated at runtime from the heap using functions like malloc, calloc, or new (in C/C++/C#/C++).

Allows for flexible allocation and deallocation of memory blocks.

Example (in C): int* dynamicArray = malloc(100 * sizeof(int));

c. Stack Allocation: Memory is allocated on the program's call stack for function calls and local variables.

Memory allocated on the stack is automatically deallocated when the function scope ends.

Used for automatic variables and function call frames.

Example: int stackVariable;

d. Pooled Allocation: Pre-allocates a pool of memory blocks of fixed sizes.

Used for managing objects or data structures that have predictable memory usage patterns.

Helps reduce memory fragmentation and overhead.

Example: Object pooling in game development for reusing frequently used objects like bullets or particles.

e. Bump Allocation: Allocates memory sequentially from a designated memory region (bump pointer).

Fast and simple allocation technique but may lead to fragmentation.

Typically used in garbage-collected environments or for short-lived objects.

B. Deallocation Techniques:

a. Manual Deallocation: Memory is deallocated explicitly by the programmer using functions like free (C), delete (C++), or Dispose (C#).

Requires careful management to avoid memory leaks and dangling pointers.

Example (in C): free(dynamicArray);

b. Reference Counting: Each object keeps track of the number of references pointing to it.

Memory is deallocated when the reference count drops to zero, indicating no active references to the object.

Used in some programming languages and libraries but may have overhead and issues with cyclic references.

c. **Pool Deallocation:** Used in pooled allocation techniques.

Memory blocks are returned to the pool for reuse after they are no longer needed.

Helps minimize allocation and deallocation overhead by reusing pre-allocated memory blocks.

d. **Scoped Deallocation:** Memory is deallocated automatically when it goes out of scope.

Commonly used in languages with automatic memory management or smart pointers (e.g., C++'s std::unique_ptr, std::shared_ptr).

7.3.2.2 Garbage collection methods (reference counting, mark-and-sweep, generational GC)

Garbage collection (GC) methods are techniques used in programming languages with automatic memory management to reclaim memory occupied by objects that are no longer in use. Here are three common garbage collection methods:

a. **Reference Counting:**

- Overview: Reference counting is a simple garbage collection technique that tracks the number of references pointing to each object.
- How it Works: Each object has a reference count, initially set to 1 for each reference. When a reference is created to the object, its count is incremented. When a reference is deleted or goes out of scope, the count is decremented. When the count reaches zero, the object is considered garbage and can be safely deallocated.
- Advantages: Immediate deallocation when the last reference is removed, minimal pause times during execution.
- Disadvantages: Inefficient for cyclic references (objects that reference each other), overhead for maintaining reference counts, difficulty in handling weak references (references that do not contribute to the count).
- Example Language: Python uses reference counting as part of its garbage collection strategy, combined with other techniques for handling cyclic references and managing memory efficiently.

b. Mark-and-Sweep:

- Overview: Mark-and-sweep is a classic garbage collection algorithm that identifies and reclaims unreachable objects by traversing the object graph.
- How it Works:
- Mark Phase: The algorithm starts from known roots (global variables, stack, registers) and traverses the object graph, marking reachable objects as live.
- Sweep Phase: Once all reachable objects are marked, the algorithm sweeps through the entire heap, deallocating memory for objects that are not marked (unreachable).
- Advantages: Handles cyclic references efficiently, works well for languages with complex object relationships, less overhead compared to reference counting.
- Disadvantages: Can cause noticeable pause times during the sweep phase, fragmentation can occur if memory is not compacted after sweeping.
- Example Language: C# and Java use variants of mark-and-sweep algorithms in their garbage collectors.

c. Generational Garbage Collection:

- Overview: Generational GC is an enhancement to mark-and-sweep that divides objects into different generations based on their age.
- How it Works:
- Young Generation: Newly created objects are placed in the young generation. A minor collection (often using copying or semi-space collection) is performed frequently on the young generation to reclaim short-lived objects.
- Old Generation: Objects that survive multiple minor collections are promoted to the old generation. A major collection (e.g., mark-and-sweep) is performed less frequently on the old generation to reclaim long-lived objects.
- Advantages: Efficient for programs with a high rate of short-lived objects (typical in many applications), reduces the overhead of full garbage collection cycles by focusing on young objects.
- Disadvantages: More complex to implement and tune, may require fine-tuning of generation sizes and collection strategies.
- Example Language: Java's HotSpot VM and .NET's CLR use generational garbage collection as part of their memory management strategies.

Each garbage collection method has its strengths and weaknesses, and the choice of method often depends on factors such as the programming language, application characteristics (memory usage patterns, object lifespan), performance requirements, and trade-offs between pause times, memory overhead, and overall system efficiency.

7.4 CALL AND RETURN MECHANISMS

The call and return mechanisms are fundamental concepts in computer programming and execution flow. Here's an overview of these mechanisms:

7.4.1 Call Mechanism

- Function Call: When a function or subroutine is called in a program, the call mechanism handles transferring control from the caller to the callee (the function being called).
- Parameters: Arguments or parameters may be passed to the function during the call, providing input data for the function's operation.
- Stack Frame: Typically, a new stack frame (activation record) is created on the program's call stack to store information such as parameters, return address, and local variables for the function call.
- Return Address: The return address is saved in the stack frame, indicating where the control flow should return after the function completes its execution.

7.4.2 Return Mechanism

- Function Execution: The callee executes its code, performing the tasks defined within the function.
- Return Value: If the function returns a value, it is computed during execution and stored in a designated location (e.g., a register or memory location) for the caller to access.
- Stack Cleanup: After the function completes execution, its stack frame is typically removed from the stack to free up memory. This process is known as stack unwinding or stack cleanup.
- Control Transfer: The return mechanism transfers control back to the caller, using the saved return address from the stack frame to resume execution at the appropriate instruction.

7.5 EXCEPTION HANDLING

In compiler design, exception handling refers to how the compiler generates code to handle exceptional conditions or errors that may occur during program execution. Here's how exception handling is typically addressed in compiler design:

a. Language Support: Many modern programming languages, especially high-level languages like Java, C#, Python, and C++, include built-in support for exception handling.

Compiler designers need to implement mechanisms to support the syntax, semantics, and runtime behavior of exception handling constructs defined by the language specification.

b. Code Generation: During the compilation process, the compiler translates high-level language constructs, including exception handling statements (e.g., try-catch blocks), into low-level code that the target platform can execute.

This involves generating instructions for throwing exceptions, catching exceptions, and handling cleanup tasks associated with exceptions.

c. Exception Propagation: When an exception occurs within a function or block of code, the compiler generates code to propagate the exception up the call stack until it is caught and handled by an appropriate catch block.

Exception propagation may involve unwinding the stack, deallocating resources, and transferring control to the nearest catch block that matches the type of the thrown exception.

d. Stack Unwinding: When an exception is thrown, the compiler generates code to unwind the call stack, deallocating resources and executing cleanup tasks as needed.

This process ensures that resources held by functions along the call chain are properly released, even if an exception interrupts the normal execution flow.

e. Exception Types and Handlers: Compiler designers must support the definition of custom exception types and the declaration of exception handlers (catch blocks) to handle specific types of exceptions.

Matching the thrown exception type to the appropriate catch block requires generating code for runtime type checking and exception dispatching.

f. Resource Management: Exception handling in compiler design often includes generating code to manage resources, such as closing files, releasing memory, or rolling back transactions, to ensure proper cleanup in the event of an exception.

g. Optimizations and Efficiency: Advanced compilers may optimize exception handling code to minimize overhead and improve runtime performance.

Techniques such as exception table optimization, lazy exception handling, and inlining of exception handling code can reduce the impact of exception handling on program execution speed.

h. Error Reporting and Debugging: Compiler-generated code for exception handling may include mechanisms for reporting error

messages, stack traces, and debugging information to aid developers in diagnosing and fixing issues related to exceptions.

Exception handling in compiler design is a complex task that involves translating high-level language constructs into efficient and reliable code for managing exceptional conditions during program execution. Compiler designers must ensure that exception handling mechanisms comply with language specifications, provide robust error handling capabilities, and optimize performance where possible.

7.6 LEXICAL AND SYNTAX ERROR HANDLING

In compiler design, handling lexical and syntax errors is crucial for producing reliable and user-friendly compilers. Here's how lexical and syntax error handling is typically addressed:

7.6.1 Lexical Error Handling

In compiler design, lexical error handling is a critical aspect of the lexical analysis phase, also known as scanning. Here's a detailed look at lexical error handling:

7.6.1.1 Introduction to Lexical Errors

Lexical errors are a type of error that occurs during the lexical analysis phase of compiling source code. This phase is also known as scanning or lexing. Lexical errors occur when the compiler encounters tokens or sequences of characters that do not conform to the language's lexical rules.

These errors typically involve invalid tokens, illegal characters, or malformed lexemes (lexical elements like identifiers, keywords, operators, and literals).

Common lexical errors (illegal characters, unclosed strings)

a. Illegal Characters:

- Definition: Illegal characters are characters that are not allowed within the syntax of the programming language. These may include non-alphanumeric characters, control characters, or characters with special meanings in the language.
- Example: Using a symbol like @ or $ in an identifier in a language that only allows letters, digits, and underscores.
- Impact: Illegal characters can lead to immediate lexical errors because they violate the language's lexical rules.

b. Unclosed Strings:

- Definition: Unclosed strings occur when a string literal in the code is not properly terminated with a closing quotation mark.
- Example: string text = "Hello, this is an unclosed string;

- Impact: Unclosed strings cause the lexer to interpret everything following the opening quotation mark as part of the string literal until it encounters the closing quotation mark or the end of the line. This can result in syntax errors or unexpected behavior in the code.

c. Mismatched Delimiters:

- Definition: Mismatched delimiters occur when pairs of delimiters (such as parentheses, braces, or brackets) are not correctly matched or nested.
- Example: if (condition { /* code block */ }
- Impact: Mismatched delimiters can lead to syntax errors or ambiguity in the code's structure. They may cause the compiler to misinterpret the intended grouping or hierarchy of code blocks, leading to compilation errors.

d. Incomplete Comments:

- Definition: Incomplete comments occur when comment delimiters (e.g., /* ... */ for block comments or // for line comments) are not properly closed.
- Example: /* This is an incomplete comment
- Impact: Incomplete comments can cause the lexer to interpret subsequent code as part of the comment, leading to unexpected behavior or compilation errors.

e. Malformed Numbers:

- Definition: Malformed numbers are numeric literals that do not adhere to the syntax rules for numbers in the programming language. This may include invalid formats, missing digits, or incorrect use of decimal points.
- Example: float number = 3.14.2; (invalid floating-point number)
- Impact: Malformed numbers can result in lexical errors or type conversion issues during compilation.

7.6.1.2 Error Recovery Strategies

Error recovery strategies in compiler design are essential for handling syntax errors and other unexpected conditions encountered during parsing. Here are two common error recovery strategies:

a. Panic Mode Recovery:

- Overview: Panic mode recovery is a robust error recovery strategy where the parser skips input tokens until it finds a designated synchronization point. Once the synchronization point is reached, parsing resumes.

- **How it Works:** When a syntax error is detected, the parser enters panic mode and discards input tokens until it finds a synchronization token or a set of tokens that can serve as a recovery point.
- The synchronization tokens are typically chosen strategically to help the parser recover and continue parsing from a known valid state.
- Example: In C-like languages, semicolons (;) are often used as synchronization points. If a syntax error is encountered, the parser may skip tokens until it finds a semicolon, indicating the end of a statement, and then resume parsing from that point.
- Advantages: Panic mode recovery is straightforward to implement and can help the parser recover from a wide range of syntax errors, allowing the compilation process to continue without halting at the first error.
- Disadvantages: It may lead to cascading errors if the parser skips over essential parts of the code, resulting in multiple error messages and potential confusion for developers.

b. Phrase Level Recovery (Local Correction):

- Overview: Phrase level recovery, also known as local correction, involves attempting to correct syntax errors within a specific phrase or production rule in the grammar.
- How it Works: When a syntax error is detected, the parser tries to identify nearby tokens that can be inserted, deleted, or substituted to transform the erroneous phrase into a valid phrase according to the grammar.
- The correction process may involve using heuristics, predictive algorithms, or predefined correction rules based on the grammar and common syntactic patterns.
- Example: If a missing semicolon is detected in a statement, the parser may attempt to insert the semicolon at the expected location to correct the error.
- Advantages: Phrase level recovery can provide more targeted and context-sensitive error correction, leading to more accurate recovery from syntax errors and potentially reducing the number of cascading errors.
- Disadvantages: It requires more sophisticated parsing techniques and error correction algorithms, making it more complex to implement compared to panic mode recovery. It may also be limited in its ability to correct certain types of errors that involve structural changes beyond the local phrase.

- Both panic mode recovery and phrase level recovery are valuable error recovery strategies in compiler design. The choice of strategy depends on factors such as the language's grammar complexity, the desired level of error correction, and the trade-offs between simplicity and accuracy in error recovery.

7.6.1.3 Error Reporting and Handling

Error reporting and handling are crucial aspects of compiler design, ensuring that developers receive clear, informative messages about errors in their code and providing mechanisms for handling and correcting those errors. Here are strategies for reporting lexical errors and techniques for handling and correcting errors in compilers:

A. Reporting Lexical Errors:

- Error Messages: When the lexer (lexical analyzer) detects a lexical error, it generates an error message to inform the developer about the nature of the error and its location in the source code.
- Error Information: Lexical error messages typically include details such as the line number, column number, the invalid token or character sequence encountered, and suggestions for correcting the error.
- Example Lexical Error Message: "Lexical error: Unexpected token '@' at line 3, column 10. Expected token: Identifier or keyword."

B. Techniques for Handling and Correcting Errors:

a. Error Recovery Strategies:

- Panic Mode Recovery: The parser skips tokens until it finds a synchronization point, such as a semicolon or a specific keyword, to resume parsing.
- Phrase Level Recovery: The parser attempts to correct syntax errors within specific phrases or production rules using heuristics or predefined correction rules.

b. Automatic Correction:

- Spell Checking: The compiler may perform basic spell checking on identifiers and keywords to detect typos or misspelled words.
- Missing Punctuation: Automatic insertion of missing punctuation, such as semicolons at the end of statements or closing braces in code blocks.

c. Interactive Suggestions:

- Code Completion: IDEs and code editors offer code completion features that suggest valid tokens, keywords, or identifiers as developers type, helping prevent lexical errors.
- Quick Fix Suggestions: IDEs provide quick-fix suggestions for common errors, allowing developers to apply corrections with a single click.

d. Syntax Highlighting and Visualization:

- Syntax Highlighting: Highlighting invalid tokens or syntax errors in the code editor helps developers identify errors visually.
- Syntax Trees: Displaying syntax trees or parse trees can help developers understand the structure of their code and identify potential errors.

e. Compiler Directives and Flags:

- Warning and Error Flags: Compiler directives allow developers to control error reporting behavior, such as treating warnings as errors or ignoring certain types of errors during compilation.
- Debugging Symbols: Including debugging symbols in compiled code helps developers trace errors back to specific source code locations during debugging.
- Effective error handling and correction in compilers enhance developer productivity, improve code quality, and facilitate the debugging process. By providing clear error messages, automated correction mechanisms, and interactive tools, compilers empower developers to write robust and error-free code more efficiently.

7.6.2 Syntax Error Handling

Syntax error handling is a crucial aspect of compiler design, focused on detecting and recovering from errors in the syntax of the programming language.

7.6.2.1 Introduction to Syntax Errors

Syntax errors are fundamental errors that occur when the compiler encounters code that does not adhere to the grammar rules of the programming language. These errors indicate deviations from the expected structure and syntax of the code, making it difficult or impossible for the compiler to interpret and generate executable code. Here's an introduction to syntax errors, including common examples such as mismatched parentheses and missing semicolons:

a. Mismatched Parentheses:

- Description: Mismatched parentheses occur when there is an imbalance between opening and closing parentheses in expressions or function calls.
- Example: if (condition) { /* code block */ (missing closing parenthesis)
- Impact: Mismatched parentheses can lead to syntax errors, as the compiler expects balanced parentheses to properly parse and interpret code blocks, conditions, and function arguments.

b. Missing Semicolons:

- Description: Missing semicolons occur when statements are not terminated with the required semicolon symbol (;) in languages that use semicolons to denote the end of statements.
- Example: int x = 10 (missing semicolon at the end of the statement)
- Impact: Missing semicolons can cause syntax errors, as the compiler interprets the absence of a semicolon as an incomplete statement, leading to unexpected behavior or compilation failures.

c. Common Syntax Errors:

- Incorrect Operator Usage: Using operators incorrectly or in unsupported contexts can result in syntax errors. For example, using arithmetic operators with non-numeric operands.
- Invalid Statement Structures: Writing statements that do not follow the language's syntax rules, such as misplaced keywords or incorrect use of control structures, can lead to syntax errors.
- Mismatched Braces or Brackets: In languages that use braces ({}) or brackets ([]) for code blocks or array indexing, mismatched or improperly nested braces or brackets can cause syntax errors.
- Incorrect Function Calls: Providing incorrect arguments or parameters in function calls, missing function declarations, or using undefined functions can result in syntax errors.
- Reserved Keywords: Using reserved keywords as identifiers or variable names can lead to syntax errors, as these keywords have specific syntactic meanings in the language.

d. Impact of Syntax Errors:

- Syntax errors prevent the compiler from generating executable code, as they indicate fundamental issues with the structure and syntax of the code.

- Fixing syntax errors requires identifying and correcting deviations from the language's grammar rules, often through careful review of error messages and code inspection.

7.6.2.2 Error Recovery Strategies

Error recovery strategies in compiler design play a crucial role in handling syntax errors and ensuring that the compilation process can continue despite encountering errors. Here's an explanation of panic mode recovery, phrase-level recovery, and error productions in grammar:

a. Panic Mode Recovery:

- Overview: Panic mode recovery is a robust error recovery strategy used by parsers to recover from syntax errors by skipping input tokens until a synchronization point is reached.
- How it Works: When a syntax error is detected, the parser enters panic mode and discards input tokens until it finds a designated synchronization token or set of tokens.
- The synchronization tokens are strategically chosen to help the parser recover and resume parsing from a known valid state.
- Example: In C-like languages, semicolons (;) are often used as synchronization points. If a syntax error occurs, the parser may skip tokens until it finds a semicolon, indicating the end of a statement, and then resume parsing from that point.
- Advantages: Panic mode recovery is straightforward to implement and can help the parser recover from a wide range of syntax errors, allowing the compilation process to continue without halting at the first error.
- Disadvantages: It may lead to cascading errors if the parser skips over essential parts of the code, resulting in multiple error messages and potential confusion for developers.

b. Phrase Level Recovery (Local Correction):

- Overview: Phrase level recovery, also known as local correction, involves attempting to correct syntax errors within a specific phrase or production rule in the grammar.
- How it Works: When a syntax error is detected, the parser tries to identify nearby tokens that can be inserted, deleted, or substituted to transform the erroneous phrase into a valid phrase according to the grammar.

  The correction process may involve using heuristics, predictive algorithms, or predefined correction rules based on the grammar and common syntactic patterns.

- Example: If a missing semicolon is detected in a statement, the parser may attempt to insert the semicolon at the expected location to correct the error.

- Advantages: Phrase level recovery can provide more targeted and context-sensitive error correction, leading to more accurate recovery from syntax errors and potentially reducing the number of cascading errors.

- Disadvantages: It requires more sophisticated parsing techniques and error correction algorithms, making it more complex to implement compared to panic mode recovery. It may also be limited in its ability to correct certain types of errors that involve structural changes beyond the local phrase.

c. Error Productions in Grammar:

- Definition: Error productions are special rules added to the grammar to handle specific types of syntax errors gracefully.

- How it Works: Error productions define how the parser should recover from known syntax errors by suggesting possible corrections or alternative valid structures.

  These productions are triggered when the parser encounters a syntax error matching the conditions specified in the error production rules.

- Example: An error production may define how to recover from a missing semicolon by inserting the semicolon and continuing parsing.

- Advantages: Error productions provide explicit guidelines for error recovery, improving the parser's ability to handle common syntax errors effectively.

- Disadvantages: Crafting error productions requires detailed knowledge of potential syntax errors and their recovery strategies, adding complexity to the grammar specification.

7.6.2.3 Error Reporting and Handling

A. Reporting Syntax Errors:

- Error Messages: When the compiler detects a syntax error during parsing, it generates an error message to inform the developer about the nature of the error and its location in the source code (line number, column).

- Error Information: Syntax error messages typically include details such as the expected token or grammar rule that was violated, the actual token found, and suggestions for correcting the error.

- Example Syntax Error Message: "Syntax error: Unexpected token '}' at line 5, column 15. Expected token: ';'"

B. Techniques for Handling and Correcting Syntax Errors:

a. Panic Mode Recovery:

- Overview: The parser skips tokens until it finds a synchronization point, such as a semicolon or a specific keyword, to resume parsing.
- Usage: Panic mode recovery is particularly effective for recovering from syntax errors that occur within code blocks or statements, allowing the compilation process to continue without halting at the first error.

b. Phrase Level Recovery (Local Correction):

- Overview: The parser attempts to correct syntax errors within specific phrases or production rules using heuristics or predefined correction rules.
- Usage: Phrase level recovery is beneficial for correcting common syntax errors such as missing semicolons, mismatched parentheses, or incorrect operator usage within expressions.

c. Automatic Correction:

- Spell Checking: The compiler may perform basic spell checking on identifiers, keywords, and syntax constructs to detect typos or misspelled words.
- Missing Punctuation: Automatic insertion of missing punctuation, such as semicolons at the end of statements or closing braces in code blocks.

d. Interactive Suggestions:

- Code Completion: Integrated Development Environments (IDEs) provide code completion features that suggest valid tokens, keywords, or syntax constructs as developers type, helping prevent syntax errors.
- Quick Fix Suggestions: IDEs offer quick-fix suggestions for common syntax errors, allowing developers to apply corrections with a single click or keystroke.

e. Syntax Highlighting and Visualization:

- Syntax Highlighting: IDEs and code editors highlight syntax errors in the code, making it easier for developers to identify and correct errors as they write code.

- Syntax Trees: Displaying syntax trees or parse trees can help developers understand the structure of their code and identify potential syntax errors.

f. Compiler Directives and Flags:

- Warning and Error Flags: Compiler directives allow developers to control error reporting behavior, such as treating warnings as errors or ignoring certain types of errors during compilation.
- Debugging Symbols: Including debugging symbols in compiled code helps developers trace errors back to specific source code locations during debugging.

By combining these techniques, compilers can effectively report syntax errors, provide guidance for error correction, and assist developers in writing syntactically correct code. IDEs and code editors further enhance the error handling experience by offering interactive tools and real-time feedback during code development.

7.7 SUMMARY

The chapter covered key concepts in compiler design, including activation records and stack management, heap memory management, call and return mechanisms, exception handling, and lexical and syntax error handling.

a. Activation Records and Stack Management:

- Activation records organize function calls and manage local variables, parameters, and return addresses.
- Stack management involves allocating/deallocating stack frames and managing pointers for function calls on the call stack.

b. Heap Memory Management:

- Heap memory allows dynamic memory allocation and includes techniques like allocation/deallocation and garbage collection methods.

c. Call and Return Mechanisms:

- Call mechanisms handle parameter passing and function call conventions, while return mechanisms manage return values and addresses.

d. Exception Handling:

- Exception handling deals with handling errors during program execution using try-catch blocks and exception propagation.

e. Lexical and Syntax Error Handling:

- Lexical error handling addresses tokenization errors, while syntax error handling deals with structural errors in the code using recovery strategies.
- These topics are fundamental to building efficient compilers and ensuring proper error handling and memory management in programming languages.

7.8 QUESTIONS FOR PRACTICE

1. Explain the structure of an activation record and its role in function/procedure call management.
2. What are the differences between stack memory and heap memory, and when would you use each?
3. Describe dynamic memory allocation techniques and compare their advantages and disadvantages.
4. How do garbage collection methods like reference counting, mark-and-sweep, and generational garbage collection work, and what are their trade-offs?
5. What are the different parameter passing methods in function calls, and how do they impact memory management and performance?
6. Discuss the concept of try-catch blocks in exception handling and explain how they help manage errors in code execution.
7. Compare panic mode recovery and phrase level recovery as error recovery strategies in compiler design. When would you use each strategy?
8. Explain the significance of stack management in compiler design, including stack frame allocation, deallocation, stack pointer, and frame pointer management.
9. How does error reporting and handling differ between lexical errors and syntax errors in compilers?
10. Describe the role of error productions in grammar and how they contribute to error recovery and correction during parsing.

7.9 REFERENCES

https://www.naukri.com/code360/library/activation-record-in-compiler-design

https://www.guru99.com/stack-vs-heap.html

https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/

https://www.dcs.warwick.ac.uk/report/pdfs/cs-rr-215.pdf

❄❄❄❄❄❄❄

8

INTRODUCTION TO COMPILER TOOLS, TECHNIQUES AND ADVANCED TOPICS IN COMPILER DESIGN

Unit Structure

8.0 Objective

8.1 Introduction

8.2 Introduction to Compiler Tools and Techniques

- 8.2.1 Overview of Compiler Design
  - 8.2.1.1 Definition and purpose of a compiler
  - 8.2.1.2 Phases of compilation
- 8.2.2 Compiler Construction Tools
  - 8.2.2.1 Lexical analyzer generators (e.g., Lex, Flex)
  - 8.2.2.2 Syntax analyzer generators (e.g., Yacc, Bison)

8.3 Lexical and Syntax Analyzer Generators

- 8.3.1 Lexical Analyzers
  - 8.3.1.1 Role of lexical analyzers in compilation
  - 8.3.1.2 Tokenization and regular expressions
- 8.3.2 Syntax Analyzers
  - 8.3.2.1 Role of syntax analyzers in compilation
  - 8.3.2.2 Context-free grammars and parsing techniques

8.4 Code Generation Frameworks

- 8.4.1 Introduction to Code Generation
  - 8.4.1.1 Objectives of code generation
  - 8.4.1.2 Intermediate representations (IR)
- 8.4.2 LLVM (Low-Level Virtual Machine)
  - 8.4.2.1 Overview of LLVM
  - 8.4.2.2 Architecture and components of LLVM
  - 8.4.2.3 Using LLVM for code generation

8.5 Debugging and Testing Compilers

- 8.5.1 Importance of Compiler Debugging and Testing
  - 8.5.1.1 Common compiler bugs and issues
  - 8.5.1.2 Strategies for debugging compilers

8.5.2 Tools and Techniques for Testing Compilers

8.5.2.1 Unit testing frameworks

8.5.2.2 Automated testing tools (e.g., Fuzzing)

8.5.2.3 Debugging tools (e.g., GDB, Valgrind)

8.6. Just-in-Time (JIT) Compilation

8.6.1 Introduction to JIT Compilation

8.6.1.1 Difference between JIT and ahead-of-time (AOT) compilation

8.6.1.2 Benefits and challenges of JIT compilation

8.6.2 JIT Compilation Techniques

8.6.2.1 Dynamic code generation

8.6.2.2 Runtime optimization strategies

8.6.3 Examples of JIT Compilers

8.6.3.1 Java HotSpot VM

8.6.3.2 .NET CLR JIT

8.7 Parallel and Concurrent Programming Support

8.7.1 Introduction to Parallel and Concurrent Programming

8.7.1.1 Importance in modern computing

8.7.1.2 Challenges in supporting parallelism and concurrency

8.7.2 Compiler Techniques for Parallelism

8.7.2.1 Automatic parallelization

8.7.2.2 Data dependence analysis

8.7.2.3 Loop transformations and optimizations

8.7.3 Tools and Frameworks

8.7.3.1 OpenMP

8.7.3.2 MPI

8.8 Compiler Optimization Frameworks

8.8.1 Introduction to Compiler Optimization

8.8.1.1 Goals and types of optimizations

8.8.1.2 Static vs. dynamic optimizations

8.8.2 Common Optimization Techniques

8.8.2.1 Loop optimizations (unrolling, fusion)

8.8.2.2 Inlining, constant folding, and dead code elimination

8.8.2.3 Register allocation and instruction scheduling

8.8.3 Optimization Frameworks

8.8.3.1 Overview of popular frameworks (e.g., LLVM's optimization passes)

8.8.3.2 How to use and extend these frameworks

8.9 Domain-Specific Language (DSL) Compilation

8.9.1 Introduction to DSLs

8.9.1.1 Definition and benefits of DSLs

8.9.1.2 Examples of domain-specific languages

8.9.2 Designing a DSL

8.9.2.1 Key considerations in DSL design

8.9.2.2 Syntax and semantics of DSLs

8.9.3 Implementing a DSL Compiler

8.9.3.1 Parsing techniques for DSLs

8.9.3.2 Code generation for specific domains

8.9.3.3 Tools and frameworks for DSL compilation (e.g., ANTLR)

8.10 Summary

8.11 Questions for Practice

8.12 References

8.0 OBJECTIVE

The primary objectives are to understand the core components of compilers, explore advanced technologies like LLVM for code generation, develop skills for debugging and testing compilers, support modern programming needs through parallel and concurrent programming, and design and implement domain-specific languages (DSLs).

8.1 INTRODUCTION

Compilers translate high-level programming languages into machine code, crucial for software development. This content covers both foundational and advanced aspects, including lexical and syntax analyzer generators, LLVM for code generation, and JIT compilation for runtime optimization. We will also explore debugging and testing best practices, support for parallel and concurrent programming, and compiler optimization frameworks. Finally, we delve into DSL compilation, enabling the creation of specialized languages tailored to specific domains.

8.2 INTRODUCTION TO COMPILER TOOLS AND TECHNIQUES

Compiler tools and techniques are essential for transforming high-level programming languages into machine code that computers can execute. This section explores the fundamental components and tools used in compiler construction, providing a comprehensive understanding of how compilers work and the technologies that support their development.

8.2.1 Overview of Compiler Design

Compiler design is a critical area of computer science that focuses on the development of compilers, which are programs that translate high-level source code into machine code, assembly language, or intermediate representations that a computer can execute. Understanding compiler design involves examining the various phases of compilation, each with distinct responsibilities and methodologies.

8.2.1.1 Definition and purpose of a compiler

A compiler is a sophisticated software tool that takes source code written in high-level programming languages (such as C, Java, or Python) and converts it into machine code, which is a low-level, binary format that the computer's processor can execute directly. The primary purposes of a compiler are:

a. Translation: Converting high-level language constructs into a form that the machine can understand and execute.

b. Optimization: Improving the efficiency of the code to ensure it runs faster and uses fewer resources.

c. Error Detection: Identifying and reporting errors in the source code to help developers correct mistakes.

d. Abstraction: Allowing programmers to write in high-level languages that are easier to understand and maintain, rather than in machine code.

8.2.1.2 Phases of compilation

The compilation process is divided into several key phases, each responsible for a specific aspect of translating and optimizing the source code:

a. Lexical Analysis:

- Purpose: The lexical analyzer (or lexer) processes the input source code to produce a sequence of tokens. Tokens are the smallest meaningful units in the code, such as keywords, operators, identifiers, and literals.

- Process: The lexer scans the source code, matching patterns defined by regular expressions to generate tokens while ignoring whitespace and comments.

b. Syntax Analysis:

- Purpose: The syntax analyzer (or parser) takes the sequence of tokens from the lexer and organizes them into a syntax tree (or parse tree) according to the grammatical rules of the programming language.

- Process: The parser checks for syntactic correctness, ensuring that the tokens form valid statements and constructs. It reports syntax errors if the structure is incorrect.

c. Semantic Analysis:

- Purpose: The semantic analyzer verifies the syntax tree for semantic correctness, ensuring that the code adheres to the rules of the language, such as type checking and variable scope.

- Process: This phase checks for logical errors and validates that operations and function calls are semantically correct.

d. Optimization:

- Purpose: The optimizer enhances the intermediate code's performance by applying various optimization techniques.

- Process: Common optimizations include constant folding, loop unrolling, dead code elimination, and inlining. The goal is to improve execution speed and reduce resource consumption.

e. Code Generation:

- Purpose: The code generator translates the optimized intermediate code into machine code or assembly language.

- Process: This phase converts high-level constructs into low-level instructions that the processor can execute, ensuring efficient use of hardware resources.

f. Code Optimization:

- Purpose: Further refine the generated machine code to enhance its performance and efficiency.

- Process: Techniques such as register allocation, instruction scheduling, and peephole optimization are applied to produce highly optimized executable code.

Understanding these phases is crucial for designing efficient and effective compilers. Each phase plays a vital role in ensuring that the source code is accurately translated and optimized, resulting in high-performance

executable programs. The next section will delve into the specific tools used in compiler construction, which automate and facilitate these processes.

8.2.2 Compiler Construction Tools

Compiler construction tools are essential for automating various phases of the compilation process, enhancing efficiency, and reducing the complexity of building compilers. These tools help in generating key components of a compiler, such as lexical analyzers and syntax analyzers, thereby streamlining the development process. Below are some of the prominent tools used in compiler construction.

8.2.2.1 Lexical analyzer generators (e.g., Lex, Flex)

Lexical analyzer generators, such as Lex and Flex, are tools designed to automate the creation of lexical analyzers (lexers). These tools allow developers to define regular expressions that describe the tokens of a programming language. The generator then produces the lexer code, which scans the source code, matches patterns, and outputs tokens.

a. Lex:

- Overview: Lex is one of the oldest and most widely used tools for generating lexical analyzers. It is traditionally used in Unix-based systems.
- Functionality: Developers write a specification file containing regular expressions and corresponding actions. Lex processes this file to produce a C source file that implements the lexical analyzer.
- Usage Example: Lex is often used in conjunction with Yacc (Yet Another Compiler Compiler) to build complete compilers.

b. Flex:

- Overview: Flex (Fast Lexical Analyzer) is an enhanced version of Lex, offering better performance and additional features.
- Functionality: Flex processes a specification file similar to Lex but generates more efficient and faster lexical analyzers. It provides improved flexibility and performance.
- Usage Example: Flex is commonly used in modern compiler projects and can be integrated with tools like Bison for syntax analysis.

8.2.2.2 Syntax analyzer generators (e.g., Yacc, Bison)

Syntax analyzer generators, such as Yacc and Bison, facilitate the creation of parsers. These tools allow developers to define the grammar of a programming language using a high-level specification language. The

generator then produces the parser code, which constructs syntax trees and checks for syntactic correctness.

a. Yacc (Yet Another Compiler Compiler):

- Overview: Yacc is a traditional tool for generating parsers from context-free grammars. It is often used in combination with Lex.
- Functionality: Developers write a specification file that defines the grammar rules and associated actions. Yacc processes this file to produce a C source file that implements the parser.
- Usage Example: Yacc is used to build parsers for various programming languages and can handle complex language constructs.

b. Bison:

- Overview: Bison is a modern and more flexible alternative to Yacc. It is compatible with Yacc grammar files but offers additional features and improvements.
- Functionality: Bison processes grammar specifications to produce efficient parsers. It supports advanced features like GLR parsing and can generate parsers in languages other than C.
- Usage Example: Bison is widely used in both academic and industrial compiler projects, providing robust and flexible parsing capabilities.

These tools significantly simplify the development of compilers by automating the generation of crucial components, allowing compiler developers to focus on higher-level design and optimization tasks. By leveraging these tools, developers can build efficient, reliable, and maintainable compilers.

8.3 LEXICAL AND SYNTAX ANALYZER GENERATORS

Lexical and syntax analyzers are fundamental components of a compiler, playing crucial roles in the translation of high-level source code into executable machine code. This section delves into the specifics of these components and the tools used to generate them.

8.3.1 Lexical Analyzers

Lexical analyzers, or lexers, are a crucial component in the early stages of the compilation process. They serve as the first line of analysis, transforming the raw source code into a structured sequence of tokens that can be more easily processed by subsequent phases of the compiler.

8.3.1.1 Role of lexical analyzers in compilation

The primary role of a lexical analyzer is to read the source code and convert it into tokens. Tokens are the smallest meaningful elements in the source code, such as keywords, operators, identifiers, and literals. This transformation facilitates the work of the syntax analyzer by reducing the complexity of the input data.

- Tokenization: The lexical analyzer scans the source code and identifies sequences of characters that match predefined patterns for various tokens.
- Whitespace and Comment Removal: Lexers typically ignore whitespace and comments, focusing only on the meaningful elements of the source code.
- Error Detection: Lexers detect illegal characters and malformed tokens, reporting lexical errors that need to be corrected before further compilation can proceed

8.3.1.2 Tokenization and regular expressions

Tokenization is the process of converting a sequence of characters into a sequence of tokens. Regular expressions are essential in defining the patterns that match different types of tokens.

- Regular Expressions: Regular expressions are formal language constructs used to specify patterns for matching character sequences. They are a powerful tool for defining the lexical structure of a programming language.
- Examples of Token Patterns:
    - Keywords: Recognized by fixed patterns, such as if, else, while, return.
    - Identifiers: Typically matched by the pattern [a-zA-Z_][a-zA-Z0-9_]*, which allows for variable names, function names, etc.
    - Literals: Numeric values, string literals, and other constant values, matched by patterns like [0-9]+ for integers or \".*?\" for strings.
    - Operators and Symbols: Patterns for operators (+, -, *, /) and punctuation (;, ,, (,)).

By automating the creation of lexical analyzers, tools like Lex and Flex help streamline the compiler development process, ensuring efficient and accurate tokenization of source code. This foundational step is critical for the subsequent stages of compilation, laying the groundwork for effective syntax analysis and beyond.

8.3.2 Syntax Analyzers

The syntax analyzer, or parser, is the second phase of the compilation process. Its primary role is to analyze the sequence of tokens generated by the lexer and build a syntax tree (or parse tree) based on the grammatical rules of the programming language.

8.3.2.1 Role of syntax analyzers in compilation

a. Syntax Tree Construction: Organizing tokens into a hierarchical structure that represents the syntactic structure of the source code.

b. Syntax Error Detection: Identifying and reporting errors in the structure of the code, such as missing semicolons or mismatched parentheses.

8.3.2.2 Context-free grammars and parsing techniques

Context-free grammars (CFGs) are used to define the syntax rules of a programming language. A CFG consists of a set of production rules that describe how tokens can be combined to form valid constructs in the language.

a. Components of a CFG:

1. Non-Terminals: Symbols that can be expanded into sequences of non-terminals and terminals.
2. Terminals: Symbols that represent actual tokens produced by the lexer.
3. Production Rules: Rules that define how non-terminals can be expanded.
4. Start Symbol: The initial non-terminal from which parsing begins.

b. Parsing Techniques:

1. Top-Down Parsing: Constructs the syntax tree from the top (start symbol) and works down to the leaves (tokens). Examples include Recursive Descent Parsing.
2. Bottom-Up Parsing: Constructs the syntax tree from the leaves (tokens) and works up to the root (start symbol). Examples include LR Parsing.

8.4 CODE GENERATION FRAMEWORKS

Code generation frameworks are essential tools in software development, particularly in compiler design and related fields. They provide a structured approach to translating high-level source code into executable machine code or intermediate representations (IR).

8.4.1 Introduction to Code Generation

Code generation is a crucial part of compiler design where source code written in a high-level programming language is translated into low-level code, such as machine code or intermediate representations (IR), that can be executed by a computer. The main objectives of code generation include producing efficient and optimized code, minimizing memory usage, and ensuring correctness and compatibility with the target platform.

8.4.1.1 Objectives of code generation

The objectives of code generation include:

a. Efficiency: Generating code that executes quickly and consumes minimal system resources.

b. Optimization: Applying various optimization techniques to improve code performance and reduce redundancy.

c. Correctness: Ensuring that the generated code behaves as expected and produces accurate results.

d. Portability: Creating code that can run on different hardware architectures and operating systems.

e. Maintainability: Writing code that is easy to understand, modify, and debug.

8.4.1.2 Intermediate representations (IR)

Intermediate representations (IR) are intermediate forms of code that are generated during the compilation process. They serve as a bridge between the high-level source code and the low-level target code. IR allows compilers to perform optimizations and transformations before generating the final executable code. Common IR formats include Abstract Syntax Trees (ASTs), Three-Address Code (TAC), Static Single Assignment (SSA) form, and LLVM IR.

8.4.2 LLVM (Low-Level Virtual Machine)

LLVM is a widely-used open-source compiler infrastructure project that provides a set of modular and reusable components for building compilers and code generation tools. It is designed to support a wide range of programming languages and target platforms.

8.4.2.1 Overview of LLVM

LLVM stands for Low-Level Virtual Machine, although it's often used beyond traditional virtual machines. It includes a suite of tools, libraries, and technologies for optimizing and generating code. LLVM's design emphasizes modularity, extensibility, and performance.

8.4.2.2 Architecture and components of LLVM

The architecture of LLVM consists of several key components:

a. Frontend: Converts source code from a high-level programming language (such as C, C++, or Swift) into LLVM IR.

b. Optimizer: Applies various optimization techniques to LLVM IR, improving code performance and efficiency.

c. Backend: Generates target-specific machine code or assembly language from optimized LLVM IR.

d. Target Description: Defines the characteristics and instructions of the target hardware platform.

e. JIT Compiler: Allows LLVM to compile and execute code at runtime, commonly used in dynamic languages and Just-In-Time (JIT) compilation scenarios.

8.4.2.3 Using LLVM for code generation

LLVM can be used for various code generation tasks, including:

a. Compilers: Building compilers for programming languages by integrating LLVM's frontend, optimizer, and backend components.

b. Code Optimization: Applying LLVM's optimization passes to improve code performance and reduce executable size.

c. JIT Compilation: Dynamically compiling and executing code at runtime, suitable for languages like Python, Ruby, and JavaScript.

d. Code Analysis: Analyzing and transforming code using LLVM's intermediate representations for static analysis and program understanding.

Overall, LLVM offers a powerful and flexible framework for code generation, optimization, and compilation, making it a popular choice in the compiler and programming language development communities.

8.5 DEBUGGING AND TESTING COMPILERS

8.5.1 Importance of Compiler Debugging and Testing

Compiler debugging and testing are crucial processes in software development, especially when working on compilers or language-related tools. They ensure the correctness, reliability, and performance of the compiler-generated code.

8.5.1.1 Common compiler bugs and issues

Common issues encountered during compiler development include:

a. Parsing Errors: Incorrect parsing of source code due to syntax errors or ambiguities.

b. Semantic Errors: Incorrect handling of type checking, symbol resolution, or scope rules.

c. Code Generation Errors: Inaccurate translation of high-level constructs to machine code or intermediate representations.

d. Optimization Issues: Unexpected behavior or performance regressions introduced by optimization passes.

e. Platform-Specific Problems: Compatibility issues on different hardware architectures or operating systems.

8.5.1.2 Strategies for debugging compilers

Effective strategies for debugging compilers include:

a. Incremental Development: Building and testing compiler components step by step to isolate and address issues early.

b. Debugging Information: Generating and utilizing debugging information in compiler output to trace code transformations and optimizations.

c. Regression Testing: Running test suites to detect regressions caused by code changes or optimizations.

d. Static Analysis Tools: Using static code analyzers to identify potential bugs, code smells, and performance bottlenecks.

e. Logging and Tracing: Adding logging and tracing mechanisms to track compiler behavior and identify problematic areas.

8.5.2 Tools and Techniques for Testing Compilers

Various tools and techniques are available for testing compilers to ensure their correctness and performance.

8.5.2.1 Unit testing frameworks

Unit testing frameworks facilitate the creation and execution of test cases for individual compiler components, such as:

a. Test Input Generation: Generating synthetic or real-world source code inputs to test parsing, type checking, and code generation.

b. Assertions and Expectations: Checking expected outputs, error conditions, and compiler behavior against predefined criteria.

c. Mocking and Stubs: Simulating dependencies or external libraries to isolate and test specific compiler functionalities.

8.5.2.2 Automated testing tools (e.g., Fuzzing)

Automated testing tools, like fuzzers, help uncover edge cases, corner cases, and vulnerabilities in compiler implementations:

a. Fuzz Testing: Injecting random or mutated inputs into the compiler to trigger unexpected behavior, crashes, or security vulnerabilities.

b. Coverage Analysis: Measuring code coverage during testing to ensure thorough testing of all compiler paths and functionalities.

c. Mutation Testing: Modifying source code or IR to assess the effectiveness of test cases in detecting compiler bugs or regressions.

8.5.2.3 Debugging tools (e.g., GDB, Valgrind)

Debugging tools assist in identifying and diagnosing compiler issues during development and testing:

a. GDB (GNU Debugger): Allowing developers to debug compiler internals, inspect memory, set breakpoints, and analyze program execution.

b. Valgrind: Detecting memory leaks, buffer overflows, and other memory-related errors in compiled programs, aiding in compiler debugging and optimization.

By incorporating these debugging and testing strategies, along with relevant tools and techniques, developers can enhance the reliability, performance, and quality of compilers and language tools.

8.6. JUST-IN-TIME (JIT) COMPILATION

In computing, just-in-time (JIT) compilation (also dynamic translation or run-time compilations) is compilation (of computer code) during execution of a program (at run time) rather than before execution. This may consist of source code translation but is more commonly bytecode translation to machine code, which is then executed directly. A system implementing a JIT compiler typically continuously analyses the code being executed and identifies parts of the code where the speedup gained from compilation or recompilation would outweigh the overhead of compiling that code.

8.6.1 Introduction to JIT Compilation

8.6.1.1 Difference between JIT and ahead-of-time (AOT) compilation

AOT (Ahead-of-Time)	JIT (Just-in-Time)
Compiles code before the Angular application is loaded in the browser.	Compiles Code during runtime when the Angular app is launched in the client's browser.

AOT (Ahead-of-Time)	JIT (Just-in-Time)
Generates a production-ready output with optimizations, ready for deployment without additional build steps.	Requires an additional build for production, potentially adding extra time to the deployment process.
AOT produces smaller bundle sizes, which means faster downloads for users.	Produces larger bundle sizes due to in-browser compilation, potentially impacting loading speed.
AOT catches and reports template errors during the compilation phase, ensuring more reliable applications with fewer runtime issues.	Identifies errors during runtime, which may lead to issues being discovered after the application is already in use.
Relatively easier for beginners due to its build-time error checking and optimized output.	Can be more complex for beginners, as errors are discovered during runtime.
Does not allow dynamic updates in production, requiring a rebuild for any changes.	Allows dynamic updates during development, making it easier to see immediate results.
Easier to debug in the development phase with early error detection during the build.	Debugging is possible during runtime, which can help identify issues when they occur.
Better compatibility with older browsers, ensuring wider accessibility.	Slightly less compatible with older browsers compared to AOT.

8.6.1.2 Benefits and challenges of JIT compilation

Benefits:

a. Performance Improvement:

- JIT compilation can optimize code during execution, allowing for performance enhancements that static compilers can't achieve.
- It enables hot spot optimization, where frequently executed paths are heavily optimized.

b. Dynamic Adaptation:

- JIT compilers can adapt to the actual runtime environment and usage patterns, optimizing code based on real-time data.

c. Cross-platform Compatibility:

- JIT allows for platform-independent intermediate code (like Java bytecode) to be executed efficiently on any platform with a compatible JIT compiler.

d. Reduced Startup Time:

- Initial startup can be faster since the whole program doesn't need to be compiled upfront; instead, parts are compiled as needed.

Challenges:

a. Complexity:

- Implementing a JIT compiler is more complex than a traditional ahead-of-time (AOT) compiler, requiring advanced techniques for runtime code analysis and optimization.

b. Memory Usage:

- JIT compilation requires additional memory to store both the compiled code and the JIT compiler itself.

c. Security Concerns:

- Since JIT compilers generate code at runtime, they can potentially introduce security vulnerabilities if not carefully managed.

d. Overhead:

- The process of JIT compilation introduces runtime overhead, which can affect the initial performance of an application.

8.6.2 JIT Compilation Techniques

8.6.2.1 Dynamic code generation

Definition:

Dynamic code generation refers to the creation of executable code at runtime. This allows for optimizations based on the current execution context, such as the specific hardware or the runtime behavior of the application.

Techniques:

a. Inline Caching:

- Optimizes method calls by caching the target address of frequently called methods.

b. Speculative Optimization:

- Assumes certain conditions based on runtime profiling and optimizes the code accordingly. If assumptions fail, deoptimization can occur.

8.6.2.2 Runtime optimization strategies

a. Adaptive Optimization:

- Continuously profiles the running application and applies optimizations to hot spots—code sections executed frequently.

b. Deoptimization:

- Reverts previously applied optimizations if they are determined to be inefficient or incorrect based on new runtime information.

c. Garbage Collection Integration:

- Works with the runtime's garbage collector to optimize memory management, reducing the impact of memory allocation and deallocation on performance.

8.6.3 Examples of JIT Compilers

8.6.3.1 Java HotSpot VM

Overview:

The HotSpot VM is the JIT compiler used by Java to translate Java bytecode into native machine code.

Features:

a. Tiered Compilation:

- Combines both an interpreter and multiple JIT compilers to balance startup time and peak performance.

b. Escape Analysis:

- Optimizes object allocation and synchronization by determining if objects can be safely allocated on the stack instead of the heap.

8.6.3.2 .NET CLR JIT

Overview:

The .NET Common Language Runtime (CLR) includes a JIT compiler that translates intermediate language (IL) code into native code for execution.

Features:

a. Code Caching: Caches JIT-compiled code to avoid recompiling methods on subsequent executions.

b. Profiling: Integrates with profiling tools to provide insights into runtime performance and apply appropriate optimizations.

8.7 PARALLEL AND CONCURRENT PROGRAMMING SUPPORT

Parallel programming involves splitting tasks to run simultaneously on multiple processors for speed and performance. Concurrent programming handles multiple tasks at overlapping times, focusing on correct interaction and efficient resource use, often through threads and synchronization mechanisms.

8.7.1 Introduction to Parallel and Concurrent Programming

Parallel programming: Executes multiple sub-tasks simultaneously on different processors to boost performance.

Concurrent programming: Manages overlapping tasks to ensure efficient and correct execution.

Applications: High-performance computing, real-time systems, web servers, gaming, and data analysis.

8.7.1.1 Importance in modern computing

Performance Improvements: Parallel and concurrent programming allow for tasks to be divided and executed simultaneously, which can significantly reduce overall execution time. This is particularly beneficial for compute-intensive applications like scientific simulations, data analysis, and complex calculations.

Scalability: By distributing workloads across multiple processors or cores, applications can scale more efficiently to handle larger datasets and a greater number of users. This is crucial for applications in cloud computing, big data processing, and web services.

Efficiency: Efficient utilization of multi-core processors and multi-processor systems can lead to better performance and energy efficiency. This is essential for both high-performance computing and everyday applications to make the best use of available hardware resources.

Real-time Processing: Many applications, such as video streaming, gaming, and high-frequency trading, require real-time processing capabilities. Parallel and concurrent programming enable these applications to meet strict timing constraints and deliver responsive performance.

8.7.1.2 Challenges in supporting parallelism and concurrency

Complexity: Writing parallel and concurrent programs is more complex than writing sequential programs. It requires managing multiple execution threads, ensuring data consistency, and handling synchronization. Debugging and testing parallel programs are also more challenging.

Race Conditions: Race conditions occur when multiple threads or processes access shared resources simultaneously, and the outcome depends

on the sequence of accesses. This can lead to unpredictable behavior and bugs that are difficult to reproduce and fix.

Deadlocks: Deadlocks occur when two or more processes are waiting indefinitely for each other to release resources, causing the entire system to halt. Proper resource management and avoiding circular dependencies are critical to prevent deadlocks.

Scalability Issues: Not all algorithms and applications scale linearly with the addition of more processors or cores. Factors such as data dependencies, communication overhead, and contention for shared resources can limit the scalability of parallel and concurrent programs.

8.7.2 Compiler Techniques for Parallelism

Compiler techniques for parallelism involve optimizing code to effectively utilize multiple processors or cores for concurrent execution. Key techniques include:

a. Automatic Parallelization: Automatically converting sequential code into parallel code.

b. Loop Unrolling: Transforming loops to increase the number of instructions executed in parallel.

c. Dependency Analysis: Identifying and resolving data dependencies to enable parallel execution.

d. Thread-Level Parallelism: Dividing tasks into threads that can run concurrently.

e. Task Scheduling: Efficiently distributing tasks across multiple processors to balance the load.

f. Vectorization: Converting operations to use SIMD (Single Instruction, Multiple Data) instructions.

g. Parallel Libraries and Frameworks: Utilizing libraries and frameworks that support parallel operations, like OpenMP and MPI.

8.7.2.1 Automatic parallelization

Definition: Automatic parallelization involves the compiler analyzing the program code to identify opportunities for parallel execution and transforming the code to exploit these opportunities without requiring manual intervention from the programmer.

Techniques:

a. Loop Parallelization:

- The compiler identifies loops where iterations are independent of each other and can be executed in parallel, transforming the loop to run across multiple threads or processors.

b. Function Parallelization:

- The compiler determines which functions or methods can be executed concurrently, especially those that do not share state or have minimal interaction, and schedules them to run in parallel.

8.7.2.2 Data dependence analysis

Definition:

Data dependence analysis identifies dependencies between different parts of a program to ensure correct execution order in parallel environments.

Types:

a. Flow Dependence (True Dependence): Occurs when one statement produces a result that a subsequent statement uses. Parallel execution must respect this order to maintain correctness.

b. Anti-dependence: Occurs when a statement reads a value that is later overwritten by another statement. The compiler must ensure the read happens before the write in parallel execution.

c. Output Dependence: Occurs when two statements write to the same memory location. Proper synchronization is required to ensure the correct final value is written.

8.7.2.3 Loop transformations and optimizations

a. Loop Unrolling: Reduces the overhead of loop control by expanding the loop body to execute multiple iterations in a single pass. This can improve performance by decreasing the number of iterations and increasing instruction-level parallelism.

b. Loop Tiling (Blocking): Divides the loop iterations into smaller blocks or tiles to improve cache performance by enhancing data locality. Each tile can be processed independently, potentially in parallel.

c. Loop Fusion: Combines adjacent loops that iterate over the same range into a single loop. This reduces loop overhead and can improve cache utilization by accessing related data in a more localized manner.

8.7.3 Tools and Frameworks

8.7.3.1 OpenMP

Overview:

OpenMP (Open Multi-Processing) is an API that provides a portable and scalable model for developing parallel applications in C, C++, and Fortran. It uses compiler directives, library routines, and environment variables to specify parallelism.

Features:

a. **Pragmas:** OpenMP uses compiler directives, known as pragmas, to indicate parallel regions in the code. These pragmas are simple annotations that guide the compiler to generate parallel code.

b. **Work-sharing Constructs:** OpenMP provides constructs like #pragma omp for to parallelize loops, #pragma omp sections to divide code into parallel sections, and #pragma omp single to specify code that should be executed by only one thread.

c. **Synchronization:** OpenMP includes mechanisms to manage synchronization, such as #pragma omp critical to define critical sections, #pragma omp atomic for atomic operations, and #pragma omp barrier to synchronize threads at specific points in the program.

8.7.3.2 MPI

Overview:

MPI (Message Passing Interface) is a standardized and portable message-passing system designed to function on parallel computing architectures. It is widely used for programming distributed memory systems.

Features:

a. **Point-to-point Communication:** MPI provides functions for direct communication between pairs of processes, such as MPI_Send and MPI_Recv, enabling explicit message passing.

b. **Collective Communication:** MPI includes collective communication operations like MPI_Bcast to broadcast a message to all processes, MPI_Scatter and MPI_Gather for distributing and collecting data, and MPI_Reduce for combining data from multiple processes.

c. **Synchronization:** MPI offers synchronization mechanisms such as barriers (MPI_Barrier) to coordinate processes and ensure all processes reach a certain point before continuing, ensuring correct execution order.

8.8 COMPILER OPTIMIZATION FRAMEWORKS

Compiler optimization frameworks automate code performance improvements. Examples include LLVM, GCC, Intel Compiler (ICC), Clang, and Microsoft Visual C++ Compiler. They offer optimizations like loop optimization, inlining, vectorization, and parallelization for efficient code execution.

8.8.1 Introduction to Compiler Optimization

Compiler optimization improves code performance by applying transformations during compilation. Techniques include constant folding, loop optimization, inlining, data flow analysis, vectorization, register

allocation, and parallelization. Optimization levels and target architectures impact trade-offs between compilation time, code size, and performance.

8.8.1.1 Goals and types of optimizations

Goals:

a. **Performance Improvement:** Optimizations aim to make the compiled code run faster by reducing the number of instructions executed, improving cache utilization, and taking advantage of modern CPU features.

b. **Code Size Reduction:** Some optimizations focus on reducing the size of the generated code, which can be critical for embedded systems and applications with limited memory.

c. **Power Efficiency:** Optimizations can also reduce the power consumption of a program, which is important for mobile and embedded devices.

d. **Maintainability and Readability:** While not always a primary goal, some optimizations strive to make the generated code easier to understand and maintain.

Types:

a. Local Optimization: Focuses on optimizing small parts of the code, typically within a single basic block.

b. Global Optimization: Extends optimization efforts across multiple basic blocks or the entire function to improve performance or reduce size.

c. Interprocedural Optimization: Analyzes and optimizes across function boundaries to improve overall program performance.

8.8.1.2 Static vs. dynamic optimizations

a. **Static Optimizations:** Performed at compile time by the compiler. These optimizations analyze and transform the code without executing it. Examples include loop unrolling, inlining, and constant folding.

- **Advantages:** Can be applied once during the compilation process, leading to a simpler runtime system.
- **Disadvantages:** May miss optimization opportunities that only become apparent at runtime.

b. **Dynamic Optimizations:**

- Performed at runtime by a Just-In-Time (JIT) compiler or a runtime optimization system. These optimizations adapt to the actual execution environment and workload.

- **Advantages:** Can optimize based on real-time information, potentially leading to better performance.
- **Disadvantages:** Introduces runtime overhead and complexity.

8.8.2 Common Optimization Techniques

8.8.2.1 Loop optimizations (unrolling, fusion)

a. **Loop Unrolling:** Reduces the overhead of loop control by executing multiple iterations of the loop in a single pass. This can increase instruction-level parallelism and improve cache performance.

b. **Loop Fusion:** Combines adjacent loops that iterate over the same range into a single loop. This reduces loop overhead and can improve data locality, leading to better cache performance.

8.8.2.2 Inlining, constant folding, and dead code elimination

a. **Inlining:** Replaces a function call with the actual body of the function. This can reduce the overhead of function calls and enable further optimizations by exposing more code to the compiler.

b. **Constant Folding:** Evaluates constant expressions at compile time and replaces them with their computed values. This reduces the number of runtime computations.

c. **Dead Code Elimination:** Removes code that does not affect the program's output, such as code that is never executed or whose results are never used. This can reduce code size and improve performance.

8.8.2.3 Register allocation and instruction scheduling

a. **Register Allocation:** Assigns variables to machine registers to minimize the number of memory accesses. Effective register allocation can significantly improve performance by reducing the need for slower memory operations.

b. **Instruction Scheduling:** Reorders instructions to avoid pipeline stalls and make better use of CPU resources. This can improve the instruction throughput of the processor.

8.8.3 Optimization Frameworks

8.8.3.1 Overview of popular frameworks (e.g., LLVM's optimization passes)

a. **LLVM:** LLVM (Low-Level Virtual Machine) is a widely used compiler infrastructure that provides a set of reusable components for building compilers. LLVM includes a rich set of optimization passes that can be applied to intermediate code representation (IR).

   - Optimization Passes: LLVM's optimization passes include various techniques such as loop unrolling, inlining, constant

folding, dead code elimination, register allocation, and more. These passes can be combined in different ways to achieve the desired level of optimization.

b. **GCC:** The GNU Compiler Collection (GCC) also provides a comprehensive set of optimization passes. GCC's optimizations can be fine-tuned using compiler flags, allowing developers to balance between compilation time and runtime performance.

8.8.3.2 How to use and extend these frameworks

a. **Using Optimization Frameworks:** Developers can use optimization frameworks like LLVM and GCC by applying predefined optimization passes. For instance, in LLVM, the opt tool can be used to run specific optimization passes on LLVM IR code.

   Example Command: opt -O2 input.ll -o output.ll applies the standard optimization level O2 to the input LLVM IR file.

b. **Extending Optimization Frameworks:** Developers can extend these frameworks by writing custom optimization passes. In LLVM, this involves subclassing the llvm::FunctionPass or llvm::ModulePass classes and implementing the required optimization logic.

Example:

Creating a new pass in LLVM involves defining the pass, registering it with the pass manager, and then integrating it into the compilation pipeline.

8.9. DOMAIN-SPECIFIC LANGUAGE (DSL) COMPILATION

DSL compilation translates code from a domain-specific language (DSL) into executable code or intermediate representations. It involves parsing, semantic analysis, code generation, optional optimization, and output generation. Challenges include balancing expressiveness and performance, integrating with host languages, and developing appropriate tooling.

8.9.1 Introduction to DSLs

8.9.1.1 Definition and benefits of DSLs

Definition: Domain-Specific Languages (DSLs) are programming languages designed for specific domains or tasks. They are tailored to express concepts and operations relevant to a particular problem domain, making them more expressive and easier to use for domain experts.

Benefits:

a. **Expressiveness:** DSLs allow developers to express domain-specific concepts and operations directly, leading to clearer and more concise code.

b. **Abstraction:** By focusing on the specific domain, DSLs can hide lower-level details, reducing complexity and making code more understandable.

c. **Productivity:** Domain experts can work more efficiently with DSLs as they are designed to match their mental models and workflows.

d. **Verification and Validation:** DSLs can enable better verification and validation of domain-specific rules and constraints, leading to more robust software.

8.9.1.2 Examples of domain-specific languages

a. **SQL (Structured Query Language):** A DSL for database queries, allowing users to specify operations like selecting, updating, and manipulating data in a database.

b. **HTML (Hypertext Markup Language):** A DSL for creating web pages, defining the structure and content of web documents using tags and attributes.

c. **Regular Expressions (Regex):** A DSL for pattern matching and text processing, enabling users to define complex search patterns.

8.9.2 Designing a DSL

Designing a DSL involves defining the domain scope, identifying user needs, creating intuitive syntax and semantics, balancing expressiveness with simplicity, integrating with IDEs, deciding on compilation or interpretation, handling errors effectively, testing and validating, providing thorough documentation and examples, and fostering community engagement.

8.9.2.1 Key considerations in DSL design

a. **Domain Understanding:** Understanding the target domain is crucial for designing an effective DSL. This includes identifying domain-specific concepts, operations, and constraints.

b. **Abstraction Level:** Determine the appropriate level of abstraction for the DSL, balancing between expressiveness and simplicity for domain users.

c. **Language Features:** Choose language features and constructs that align with the domain's semantics, making it easier for users to write and understand DSL code.

d. **Tooling and Integration:** Consider tooling support and integration with existing development environments to enhance the usability and adoption of the DSL.

8.9.2.2 Syntax and semantics of DSLs

a. **Syntax:** Define the syntax of the DSL using a formal notation such as BNF (Backus-Naur Form) or EBNF (Extended Backus-Naur Form). This specifies the grammar rules for valid DSL expressions.

b. **Semantics:** Define the semantics of DSL constructs, including their behavior, effects, and interactions. This clarifies how DSL code is interpreted and executed.

8.9.3 Implementing a DSL Compiler

Implementing a DSL compiler involves:

a. Tokenizing and parsing DSL code into a syntax tree.

b. Validating syntax and semantics, resolving identifiers, and detecting errors.

c. Generating executable code or intermediate representations.

d. Integrating with tooling, testing, debugging, optimizing, documenting, and deploying the compiler for distribution.

8.9.3.1 Parsing techniques for DSLs

a. **Lexer and Parser:** Use lexer and parser generators like ANTLR, Yacc, or Bison to parse DSL code and generate an abstract syntax tree (AST) representing the code's structure.

b. **Semantic Analysis:** Perform semantic analysis on the AST to check for correctness, resolve references, and enforce domain-specific rules and constraints.

8.9.3.2 Code generation for specific domains

a. **Intermediate Representation (IR):** Translate the AST into an intermediate representation suitable for code generation. This IR captures the semantics of DSL constructs in a form that can be transformed into executable code.

b. **Code Generation:** Generate target code (e.g., machine code, bytecode, or source code in another language) based on the IR. This step translates DSL constructs into executable instructions or operations.

8.9.3.3 Tools and frameworks for DSL compilation (e.g., ANTLR)

a. **ANTLR (ANother Tool for Language Recognition):** ANTLR is a powerful parser generator that can be used to create parsers and translators for DSLs. It supports various target languages and provides tools for syntax highlighting, code generation, and error handling.

b. **Other Tools:** Other tools and frameworks like JetBrains MPS (Meta Programming System), Xtext, and Spoofax can also be used for DSL

development and compilation, offering different features and capabilities for DSL designers and implementers.

8.10 SUMMARY

- Compiler Tools and Techniques

  Overview of Compiler Design and its phases.

  Compiler Construction Tools like Lex, Yacc, and Bison.

- Lexical and Syntax Analyzer Generators

  Role of lexical and syntax analyzers in compilation.

  Generators like Lex, Flex, Yacc, and Bison.

  Tokenization, regular expressions, and parsing techniques.

- Code Generation Frameworks

  Introduction to Code Generation and LLVM.

  Intermediate Representations (IR) and LLVM architecture.

- Debugging and Testing Compilers

  Importance, strategies, and tools for debugging and testing compilers.

- Just-in-Time (JIT) Compilation

  Benefits/challenges, techniques, and examples of JIT Compilers.

- Parallel and Concurrent Programming Support

  Importance, challenges, and Compiler Techniques for Parallelism.

  Tools and Frameworks like OpenMP and MPI.

- Compiler Optimization Frameworks

  Goals, types, and common techniques of optimizations.

  Overview of popular frameworks like LLVM's optimization passes.

- Domain-Specific Language (DSL) Compilation

  Introduction to DSLs, benefits, and examples.

Design considerations, syntax, semantics, and DSL Compiler implementation.

8.11 QUESTIONS FOR PRACTICE

1. What are the key phases in compiler design, and what is the purpose of each phase?
2. How do lexical analyzers and syntax analyzers contribute to the compilation process?

3. Can you explain the role of intermediate representations (IR) in code generation?
4. What are some common techniques used in compiler optimization, and how do they improve code performance?
5. What are the benefits and challenges of Just-in-Time (JIT) compilation compared to ahead-of-time (AOT) compilation?
6. Describe the importance of parallel and concurrent programming support in modern computing, and discuss some challenges in achieving parallelism.
7. How do tools like OpenMP and MPI aid in parallel programming, and what are their key features?
8. What are the objectives of code generation, and how does LLVM contribute to this process?
9. What are domain-specific languages (DSLs), and what are the benefits of using DSLs for specific tasks?
10. Explain the key considerations in designing a DSL and implementing a DSL compiler, including parsing techniques and code generation for specific domains.

8.12 REFERENCES

https://www.prepbytes.com/blog/computer-fundamentals/phases-of-a-compiler/

https://tinman.cs.gsu.edu/~raj/4330/slides/c04.pdf

https://www.monarch-innovation.com/aot-vs-jit-compiler-in-angular#:~:text=There%20are%20two%20main%20options,larger%20projects%20or%20production%20environments.

https://oxylabs.io/blog/concurrency-vs-parallelism

https://subscription.packtpub.com/book/programming/9781782160304/1/ch01lvl1sec09/implementing-a-dsl

❄❄❄❄❄❄❄