(PCCS4305)

COMPILER DESIGN

KISHORE KUMAR SAHU

SR. LECTURER, DEPARTMENT OF INFORMATION TECHNOLOGY

ROLAND INSTITUTE OF TECHNOLOGY, BERHAMPUR

PCCS4305 Compiler Design (3-0-0)

MODULE – 1 **(Lecture hours: 13)**

Introduction: Overview and phases of compilation. **(2-hours)**

Lexical Analysis: Non-deterministic and deterministic finite automata (NFA & DFA), regular grammar, regular expressions and regular languages, design of a lexical analyser as a DFA, lexical analyser generator. **(3-hours)**

Syntax Analysis: Role of a parser, context free grammars and context free languages, parse trees and derivations, ambiguous grammar.

Top Down Parsing: Recursive descent parsing, LL(1) grammars, non-recursive predictive parsing, error reporting and recovery.

Bottom Up Parsing: Handle pruning and shift reduces parsing, SLR parsers and construction or SLR parsing tables, LR(1) parsers and construction of LR(1) parsing tables, LALR parsers and construction of efficient LALR parsing tables, parsing using ambiguous grammars, error reporting and recovery, parser generator. **(8-hours)**

MODULE – 2 **(Lecture hours: 14)**

Syntax Directed Translation: Syntax directed definitions (SDD), inherited and synthesized attributes, dependency graphs, evaluation orders for SDD, semantic rules, application of syntax directed translation. **(5-hours)**

Symbol Table: Structure and features of symbol tables, symbol attributes and scopes. **(2-hours)**

Intermediate Code Generation: DAG for expressions, three address codes - quadruples and triples, types and declarations, translation of expressions, array references, type checking and conversions, translation of Boolean expressions and control flow statements, back patching, intermediate code generation for procedures. **(7-hours)**

MODULE – 3 **(Lecture hours: 8)**

Run Time Environment: storage organizations, static and dynamic storage allocations, stack allocation, handlings of activation records for calling sequences. **(3-hours)**

Code Generations: Factors involved, registers allocation, simple code generation using stack allocation, basic blocks and flow graphs, simple code generation using flow graphs. **(3-hours)**

Elements of Code Optimization: Objective, peephole optimization, concepts of elimination of local common sub-expressions, redundant and un-reachable codes, basics of flow of control optimization. **(2-hours)**

Text Book:

Compilers – Principles, Techniques and Tools, Authors: Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman , Publisher: Pearson

Lecture Manual

Lecture 01: Overview of Compiler
Lecture 02: Phases of compilation.
Lecture 03: Non-deterministic and deterministic finite automata (NFA & DFA),
Lecture 04: Regular grammar, regular expressions and regular languages.
Lecture 05: Design of a lexical analyser as a DFA, lexical analyser generator.
Lecture 06: Role of a parser, context free grammars and context free languages, parse trees and derivations, ambiguous grammar.
Lecture 07: *Top Down Parsing:* Recursive descent parsing, LL(1) grammars,
Lecture 08: Non-recursive predictive parsing, error reporting and recovery.
Lecture 09: *Bottom Up Parsing*: Handle pruning and shift reduces parsing, SLR parsers and construction or SLR parsing tables,
Lecture 10: LR(1) parsers and construction of LR(1) parsing tables,
Lecture 11: LALR parsers and construction of efficient LALR parsing tables,
Lecture 12: Parsing using ambiguous grammars,
Lecture 13: Error reporting and recovery, parser generator.
Lecture 14: Syntax directed definitions (SDD),
Lecture 15: Inherited and synthesized attributes,
Lecture 16: Dependency graphs,
Lecture 17: Evaluation orders for SDD,
Lecture 18: Semantic rules, application of syntax directed translation.
Lecture 19: Structure and features of symbol tables,
Lecture 20: Symbol attributes and scopes.
Lecture 21: DAG for expressions,
Lecture 22: Three address codes - quadruples and triples,
Lecture 23: Types and declarations, translation of expressions,
Lecture 24: Array references, type checking and conversions,
Lecture 25: Translation of Boolean expressions and control flow statements,
Lecture 26: Back patching,
Lecture 27: Intermediate code generation for procedures.
Lecture 28: Storage organizations, static and dynamic storage allocations,
Lecture 29: Stack allocation,
Lecture 30: Handlings of activation records for calling sequences.
Lecture 31: Factors involved in code generation, registers allocation,
Lecture 32: Simple code generation using stack allocation,
Lecture 33: Basic blocks and flow graphs, simple code generation using flow graphs.
Lecture 34: Objective, peephole optimization, concepts of elimination of local common sub-expressions, redundant and un-reachable codes,
Lecture 35: Basics of flow of control optimization.

CHAPTER-01
INTRODUCTION TO COMPILERS

WHY TO STUDY COMPILER DESIGN

- To study the constructs of modern programming languages and their implementation in machine language of a typical computer.
- To study the development of tools that can be used in the construction of certain translator components.

OVERVIEW OF COMPILERS

Translator

A translator is a program that takes as input a program written in one programming language i.e. *source language* and produces as output a program in another language i.e. *object or target language*.

Compiler

If the source language is a high-level language like C, C++ etc and the object language is a low-level language like assembly or machine language, then such a translator is called a compiler.

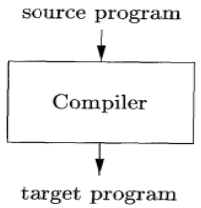

Figure 1.1: A compiler

Executing a program written in high-level programming language is basically a two step process:

- The source program must first be compiled that is translated into the object program.
- Object program is loaded into memory and executed.

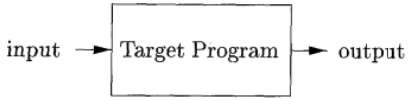

Figure 1.2: Running the target program

Interpreter

Translator that transforms a programming language into a simplified language called *intermediate code*, which can be directly executed using a program called an *interpreter.* E.g. BASIC, SNOBOL, JCL etc.

It is smaller than compilers but suffers with the disadvantage that, it is slower in execution of intermediate code that the object program in case of compiler.

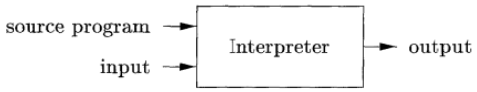

Figure 1.3: An interpreter

Hybrid Compiler

Some languages like JAVA make use of compiler as well as interpreter. First the source code is converted to *byte code* by means of a translator i.e. compiler and then a virtual machine i.e. interpreter runs the bytecode to give the output. The compiler used over here are called as hybrid compilers.

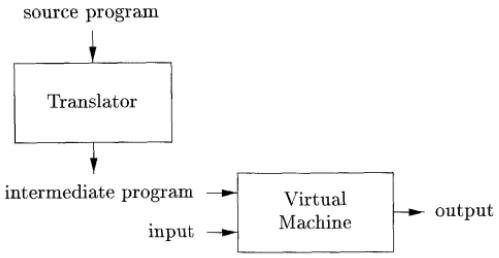

Figure 1.4: A hybrid compiler

Assembler

If the source language is assembly language and the target language is machine language, then the translator is called an assembler.

Preprocessor

If the source language is a high-level language and the target language is a different high-level language then the translator is called as preprocessor. E.g. FORTRAN preprocessor.

NEED OF TRANSLATORS

Programming in machine language is difficult because we communicate directly with computer in terms of bits, registers, and very primitive machine operations. Since code in machine language is combinations of 0's and 1's, so it is difficult to differentiate between operator and operands. Hence it is impossible to modify the codes in machine language. Some more reasons for the need of translators are as follows:

- ***Symbolic Assembly Language :*** Assembly language is a symbolic high-level language that uses mnemonics names for operator and operands. The computer cannot understand these mnemonics; hence a translator is required to translate these symbols to binary code. This translator is called as assembler.
- ***Macros :*** Assembly language has feature called macro that represents text replacement capability. In case of macro, the assembly codes are substituted for a macro name. It means, without which the code represented by the macro need to inserted wherever necessary. So the *macro processor* replaces the macro body with appropriate arguments whenever invoked.
- ***High-Level Languages :*** It is difficult to program in assembly language also, as the programmer must be aware with the instruction formats, data representation and complex operand mnemonics. To avoid these

disadvantages, high-level languages has been developed that allows a programmer to express algorithms in a more natural notation that avoids many of the details of how a specific computer functions. But it is far from being understood by the computer. Hence we have a compiler, yet another program that helps in translating the high-level language to machine language. A compiler in complex in structure than assembler.

STRUCTURE OF A COMPILER

The basic function of a compiler to translate the high-level code to machine instruction, which can never be thought as a single step process. To avoid complexity in the structure of a compiler, it is divided into series of sub processes called *phases. A phase is a logical cohesive operation that takes as input one representation of the source program and produces as output another representation.*

The figure shows the phases of a compiler. The work of each phase is as follows:

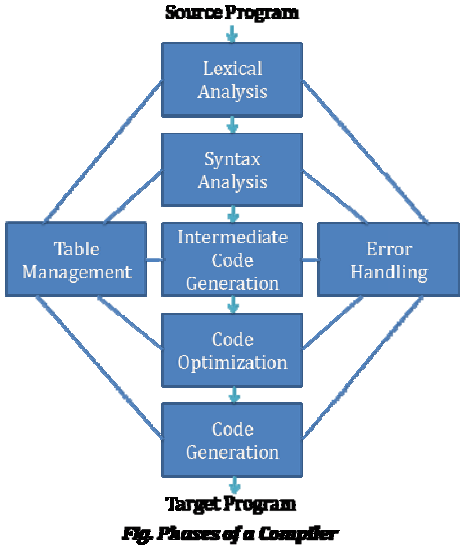

Fig. Phases of a Compiler

Lexical Analysis: Otherwise called as *scanner* is the first phase of compilation. It separates characters of the source code into groups that logically belongs together; these groups are called tokens. The usual tokens are keywords, identifier, operators etc. The set of tokens generated by the scanner is passed on to the next phase with other relevant information.

Syntax Analysis: Otherwise called as parser is the second phase of compilation. It groups tokens together into syntactic structures like expressions i.e. A+B, that could be further combined to form statements. Often the syntactic structure is represented as a parse tree whose nodes are the token form the scanner.

Semantic Analysis: After the parse tree has been generated, the types of the operators in an expressions are checked. This is called as *type checking*. If there is a mismatch in the type then the data type are same by converting the lower data type to the higher one, this is called as *coercion*.

Intermediate Code Generation: Accepts syntactic structures form the parser and produces stream of simple instructions. There are many ways of representation of intermediate code but the common is one that uses one operator and few operands. Intermediate code are similar to that of the assembly code but differ in a way that they do not specify the register to be used for each operation.

Code Optimization: An optional phase designed to improve the intermediate code so that the ultimate object program runs faster and/or takes less space. The output is another intermediate code that does the same job as the original but perhaps in a way that saves time and/or space.

Code Generation: Last phase of compilation that produces the object code by deciding on the memory locations for data, selecting code to access each datum, and selecting the registers in which each computation is to be done. Designing an effective code generator is a difficult part of compiler design, both practically and theoretically.

Table Management: Otherwise called *bookkeeping*, that keeps track of the names used by the program and records essential information about each, such as its type (integer, real, etc). The data structure used to record this information is called as *symbol table*.

Error Handler: All the phases are linked to error handler as error may be encountered at any phase in the process of compilation. These errors may be like

- Lexical error due to a misspelled token
- Syntax error due to a sentence not in accordance with the programming language syntax.
- Intermediate code generator error due to mismatch in the type of the operands
- Code optimization error due to a code being unreachable.
- Code generation error due to a compiler generated constant, too big to fit into a computer word.
- Symbol table entry error due to multiple entry of an identifier.

Passes: Portions of one or more phases are combined into a module called a pass. A pass reads the source program or the output of the previous pass, makes the transformations specified by its phases, and writes output into an intermediate file, which may then be read by a subsequent pass.
The number of passes and the grouping of phases depends on the language structure and machine construction also. A minimum of two passes is required to compile most of the language codes as they allow use of a variable before its declaration. A *multi-pass* compiler is preferred on machine that has less memory and is even slower in execution than that of a single pass compiler that runs on machine having large memory and faster in execution. This is because a multi-pass compiler reads and writes an intermediate file.

The reduction in number of passes can be accomplished by a technique called as *backpatching*. During the compilation process, the output of some phases cannot be determined as they depend on the inputs of the later phases. To overcome this huddle the phase can generate output with *slots* that can be filled later after more the input can be read. E.g. GOTO statement implementation in case of a single pass compiler we make use of backpatching technique.

COMPILER WRITING TOOLS

There are a number of tools that are used to construct a compiler called as *compiler-compiler, compiler-generator* or *translator-writing system*. They produce compiler form some form of specification of a source language and target machine. The input specification for these systems may contain:

- A description of the lexical and syntactic structure of the source language.
- A description of what output is to be generated for each source language construct, and
- A description of the target machine.

There is a trade-off between how much of the work the compiler-compiler can do automatically for the user to the flexibility of the system. E.g. the rules for identifier name is usually same for all the compiler, but if the language is flexible enough to add more rule for naming identifier then very less can be done by the compiler-complier.
The supports given by compiler-compiler are as follows:

- *Scanner generator:* A scanner can be most easily designed by giving the corresponding regular expressions for the tokens.

- *Parser generator:* A description of the syntax used in the language can be made input to the compiler-compiler in the form of context free grammar to obtain a parser. A parser is a unique phase in the compiler design and hence a mechanically generated compiler is more reliable that produced by hand.

- *Syntax-directed translation engines*: that produce collections of routines for walking a parse tree and generating intermediate code.

- *Code-generator generators*: that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.

- *Data-flow analysis engines*: that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.

- *Compiler-construction toolkits*: that provide an integrated set of routines for constructing various phases of a compiler.

- *Facilities for the code generation:* The mapping of the high-level language to that of the assembly, intermediate or object code is provided to the compiler-compiler, so the routine may be called at the correct time in generation. They also specifies the decision table that select the object code.

BOOTSTRAPPING

Three languages characterize a compiler are as follows:

- The source language (X)

- The target language (Y)

- The language in which the compiler is written (Z)

The compilers are represented as C_Z^{XY}, where X, Y and Z are as above. It is also possible to produce object code for a different machine while running on a machine. This phenomenon is called as *cross-compiler*.

Bootstrapping is process of creation of a new compiler for a different machine by making use of existing compiler. E.g. let take us have two machine A and B. And we are to create a compiler for machine B by making use of a compiler for machine A.

C_S^{LA} ------>|C_A^{SA} |--------->C_A^{LA} //Compiler for machine A written in A for language L.

C_L^{LB} ------>|C_A^{LA} |--------->C_A^{LB} //Compiler for machine B written in A for language L.

C_L^{LB} ------>|C_A^{LB} |--------->C_B^{LB} //Compiler for machine B written in B for language L.

The above equations are based on the principle can be listed as below:

C_A^{XY} ------>|C_A^{AB} |--------->C_B^{XY}

This is how we obtain the compiler for a new machine form an existing compiler of a different machine.