© 2014 IJIRT | Volume 1 Issue 5 | ISSN : 2349-6002

# Research paper on Compiler Design

Jatin Chhabra, Hiteshi Chopra, Abhimanyu Vats
*Student (B.Tech 7th sem), Department of Computer Science and Enginering*
*Dronacharya Colege of Enginering, Gurgaon, India*

***Abstract-*** **A compiler translates and/or compiles a program written in a suitable source language into an equivalent target language through a number of stages. Starting with recognition of token through target code generation provide a basis for communication interface between a user and a processor in significant amount of time. A new approach GLAP model for design and time complexity analysis of lexical analyzer is proposed in this paper. In the model different steps of tokenizer (generation of tokens) through lexemes, and better input system implementation have been introduced. Disk access and state machine driven Lex are also reflected in the model towards its complete utility. The model also introduces generation of parser. Implementation of symbol table and its interface using stack is another innovation of the model in acceptance with both theoretically and in implementation widely. The course is suitable for advanced undergraduate and beginning graduate students. Auxiliary tools, such as generators and interpreters, often hinder the learning: students have to fight tool idiosyncrasies, mysterious errors, and other poorly educative issues. We introduce a set of tools especially designed or improved for compiler construction educative projects in C.We also provide suggestions about new approaches to compiler Construction. We draw guidelines from our experience to make tools suitable for education purposes. The final result of this paper is to provide a general knowledge about compiler design and implementation and to serve as a springboard to more advanced courses.**

## I. INTRODUCTION

Compilers and operating systems constitute the basic interfaces between a programmer and the machine. Compiler is a program which converts high level programming language into low level programming language or source code into machine code. Understanding of these relationships eases the inevitable transitions to new hardware and programming languages and improves a person's ability to make appropriate trade off in design and implementation. Many of the techniques used to construct a compiler are useful in a wide variety of applications involving symbolic data. The term compilation denotes the conversion of an algorithm expressed in a human-oriented source language to an equivalent algorithm expressed in a hardware-oriented target language. We shall be concerned with the engineering of compilers their organization, algorithms, data structures and user interfaces.It is not difficult to see that this translation process from source text to instruction sequence requires considerable effort and follows complex rules. The construction of the first compiler for the language Fortran(formula translator) around 1956 was a daring enterprise, whose success was not at all assured. It involved about 18 man years of effort, and therefore figured among the largest programming projects of the time. Programming languages are tools used to construct formal descriptions of finite computations (algorithms). Each computation consists of operations that transform a given initial state into some final state.

## II. STORAGE MANAGEMENT

In this section weshall discuss management of storage for collections of objects, including temporary variables,during their lifetimes. The important goals are the most economical use of memory and thesimplicity of access functions to individual objects. Source language properties govern thepossible approaches, as indicated by the following questions :
1. Is the extent of an object restricted, and what relationships hold between the extentsof distinct objects (e.g. are they nested)?
2. Does the static nesting of the program text control a procedure's access to global objects,or is access dependent upon the dynamic nesting of calls?
3 Is the exact number and size of all objects known at compilation time?

- Frontend
  - Dependent on source language
  - Lexical analysis
  - Parsing
  - Semantic analysis (e.g., type checking)

**Static Storage Management**

We speak of static storage management if the compiler can provide fixed addresses for allobjects at the time the program is translated (here we assume that translation includesbinding), i.e. we can answer the first question above with 'yes'. Arrays with dynamic bounds,recursive procedures and the use of anonymous objects are prohibited. The condition is fulfilled for languages like FORTRAN and BASIC, and for the objects lying on the outermostcontour of an ALGOL 60 or Pascal program. (In contrast, arrays with dynamic bounds canoccur even in the outer block of an ALGOL 68 program.)If the storage for the elements of an array with dynamic bounds is managed separately,the condition can be forced to hold in this case also.

**Dynamic Storage Management Using a Stack**

All declared values in languages such as Pascal andSIMULA have restricted lifetimes. Further, the environments in these languages are nested:The extent of all objects belonging to the contour of a block or procedure ends before that ofobjects from the dynamically enclosing contour. Thus we can use a stack discipline to managethese objects: Upon procedure call or block entry, the activation record containing storage forthe local objects of the procedure or block is pushed onto the stack. At block end, procedurereturn or a jump out of these constructs the activation record is popped of the stack. (Theentire activation record is stacked, we do not deal with single objects individually!)An object of automatic extent occupies storage in the activation record of the syntacticconstruct with which it is associated. The position of the object is characterized by the baseaddress, b, of the activation record and the relative location offset), R, of its storage withinthe activation record. R must be known at compile time but b cannot be known (otherwisewe would have static storage allocation). To access the object, b must be determined at runtime and placed in a register. R is then either added to the register and the result usedas an indirect address, or R appears as the constant in a direct access function of the form'register+constant'.The extension, whichmay vary in size from activation to activation, is often called the second order storage of theactivation record.

## III. Error Handling

Error Handling is concerned with failures due to many causes: errors in the compiler or itsenvironment (hardware, operating system), design errors in the program being compiled, anincomplete understanding of the source language, transcription errors, incorrect data, etc.The tasks of the error handling process are to detect each error, report it to the user, andpossibly make some repair to allow processing to continue. It cannot generally determinethe cause of the error, but can only diagnose the visible symptoms. Similarly, any repaircannot be considered a correction (in the sense that it carries out the user's intent); it merelyneutralizes the symptom so that processing may continue. The purpose of error handling is to aid the programmer by highlighting inconsistencies.It has a low frequency in comparison with other compiler tasks, and hence the time requiredto complete it is largely irrelevant, but it cannot be regarded as an 'add-on' feature of acompiler. Its inuence upon the overall design is pervasive, and it is a necessary debuggingtool during construction of the compiler itself. Proper design and implementation of an errorhandler, however, depends strongly upon complete understanding of the compilation process.This is why we have deferred consideration of error handling until nowErrors, Symptoms, Anomalies and LimitationsWe distinguish between the actual error and its symptoms. Like a physician, the error handlersees only symptoms. From these symptoms, it may attempt to diagnose the underlyingerror. The diagnosis always involves some uncertainty, so we may choose simply to report thesymptoms with no further attempt at diagnosis.

## IV. The Structure Of A Compiler

1. A simple expression language
2. Loops and conditionals
3. Functions
4. Structs and arrays
5. Memory safety and basic optimizations

## V. Compiler Requirements

**Interoperability**- Programs do not run in isolation, but are linked withlibrary code before they are executed, or will be called as a library fromother code. This puts some additional requirements on the compiler.

**Efficiency**- The early emphasis on correctness has consequences for your approachto the design of the implementation. Modularity and simplicity of the codeare important for two reasons: first, your code is much more likely to becorrect, and, second, you will be able to respond to changes in the sourcelanguage specification.In a production compiler, efficiency of the generated code andalso efficiency of the compiler itself are important considerations. In thiscourse, we set very lax targets for both, emphasizing correctness instead. Inone of the later labs in the course, you will have the opportunity to optimizethe generated code.

## REFERENCES

[1] Aho, Alfred V., Hop croft, J. E., and Ullman, Jeffrey D. [1974]. The Design andAnalysis of Computer Algorithms.Addision Wesley, Reading, MA.

[2] William M. WaiteDepartment of Electrical EngineeringUniversity of ColoradoBoulder, Colorado 80309USAemail: William.Waite@colorado.edu.

[3]GerhardGoosInstitutProgrammstrukturen und DatenorganisationFakultat fur Informatik

[4] Aho, Alfred V. and Johnson, Stephen C. [1976]. Optimal code generation for expression trees. Journal of the ACM, 23(3):488501.

[5] Ross, D. T. [1967]. The AED free storage package. Communications of the ACM, 10(8):481492.

[6] Rutishauser, H. [1952]. Automatische Rechenplanfertigung bei Programm-gesteuerten

[7] Niklaus WirthThis is a slightly revised version of the book published by Addison-Wesley in 1996ISBN 0-201-40353-6Zürich, November 2005.

[8] Aho, Alfred V. and Ullman, Jeffrey D. [1972]. The Theory of Parsing, Translation, [9] Aho, Alfred V. and Ullman, Jeffrey D. [1977]. Principles of Compiler Design.Addision