Chronological Advancement in Compiler Design: A Review

Amit Verma[1] and Nikita Bakshi[2]

Department of Computer Science and Engineering,
Chandigarh University, Mohali, India.
[1]amit.verma@cumail.in
[2]bakshinikita24@gmail.com

Abstract. Compiler is a set of instructions that translate the source code into binary format usually known as object code. Compiler is used to convert a language readable in user domain into the tasks which is understood by the machine. For example C++ compiler compiles program written in a language easily understandable by human which compiles task that can be executed by a computer's processor. In this paper, we discuss about the computers and their evolution. Algorithms and tools are used for compiler design. Further our study carrys a survey on key properties of compiler courses in some universities.

Keywords: Computer, Compiler, Programming language, Memory Aware Mapping, Automatic Parallelizing, Lemon.

1 Introduction

Computer is an electronic device which is competent of receiving information in a particular form and performing a set of instructions with predetermined but variable set of instruction to produce a result in the form signals. Sequence of operations can be easily changed in computer so it can solve more than one kind of problem like logical operations etc.

1.1 Evolution of Computers

1822: English mathematician Charles Babbage conceives of a steam-driven calculating machine that would be able to compute tables of numbers. The project, funded by the English government, is a failure. More than a century later, however, the world's first computer was actually built. [15]

1890: Herman Hollerith designs a punch card system to calculate the 1880 census, accomplishing the task in just three years and saving the government $5 million. He establishes a company that would ultimately become IBM (IBM was founded in 1911). [13]

1937: J.V. Atanasoff, a professor of physics and mathematics at Iowa State University, attempts to build the first computer without gears, cams, belts or shafts. [14]

1941: Atanasoff and his graduate student, Clifford Berry, design a computer that can solve 29 equations simultaneously. This marks the first time a computer is able to store information on its main memory. [15]

1943–1944: Two University of Pennsylvania professors—John Mauchly and J. Presper Eckert—build the Electronic Numerical Integrator and Calculator (ENIAC). Considered the grandfather of digital computers, it fills a 20 foot by 40 foot room and has 18,000 vacuum tubes. [16]

1946: Mauchly and Presper leave the University of Pennsylvania and receive funding from the Census Bureau to build the UNIVAC, the first commercial computer for business and government applications. [16]

1953: Grace Hopper develops the first computer language, which eventually becomes known as COBOL. Inventor Thomas Johnson Watson, of IBM CEO Thomas Johnson Watson, conceives the IBM 701 EDPM to help the United Nations keep tabs on Korea during the war. [15]

1954: The FORTRAN programming language came into existence.

1960–1962: In 1960, COBOL became an early high-level programming to be compiled on different architectures. In 1962, the first self-hosting compiler was assigned for Lisp by Tim Hart and Mike Levis at MIT. [15]

1980–1999: The term **Wi-Fi** becomes part of the computing language and users begin connecting to the Internet without wires.

2000–2015: Apple unveils the **iPad,** changing the way consumers view media and jumpstarting the dormant tablet computer segment.

1.2 Compiler

Compiler is a program that translates source code into object code. The compiler derives its name from the way it works, looking at the entire piece of source code and collecting and reorganizing the instructions. Thus, a compiler differs from an interpreter, which analyzes and executes each line of source code in succession, without looking at the entire program. [17]

1.3 Phases of Compiler

A compiler efficiently generates object code by confirming code syntax. At run time, output is formatted according to the rules of linker and assembler. A compiler consists of:

a. Lexical Analysis: Lexical analysis is used to remove irrelevant information from the program source. Irrelevant information contains things like blanks and comments. Besides eliminating irrelevant information, lexical analysis determines the lexical tokens of the language.

b. Syntax Analysis: Syntax Analysis is responsible for looking syntax rules of the language (often as a context-free grammar), and construction of an intermediate representation of the language.

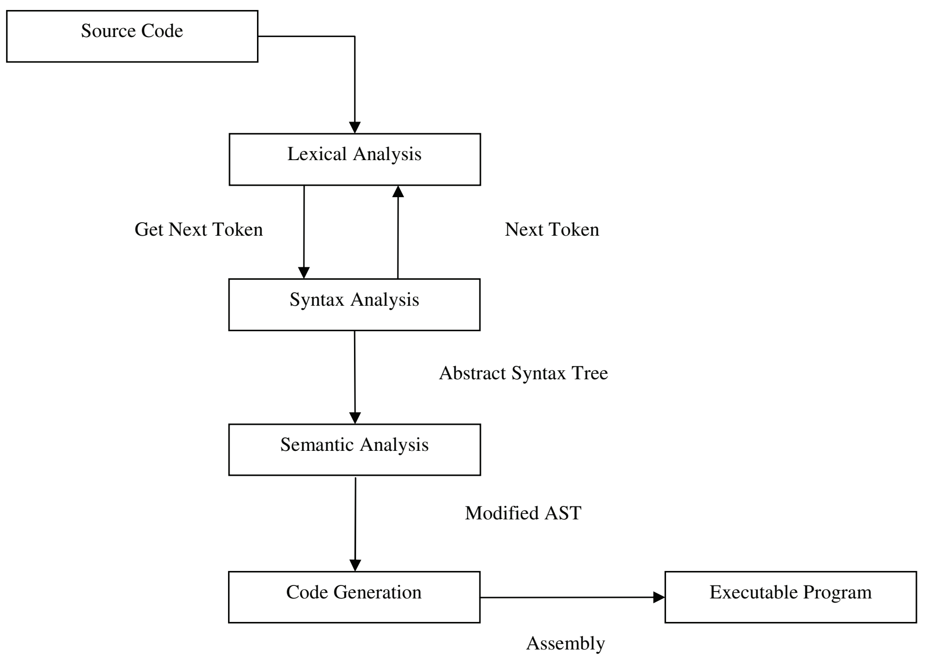

Fig. 1. Phases of Compiler.

c. Semantic Analysis: Semantic analysis takes the representation made from the analysis of syntax and semantic rules apply for representation to ensure that the program meets the requirements of semantic rules of the language.

d. Code Generation: This final stage of a typical compiler converts the intermediate representation of program into an executable set of instructions. This last stage is the only step in the compilation that is machine dependent. You can also do optimization at this stage of compilation that makes the program more efficient.

This paper is organized as follows: Section II provides the related work. In Section III, we briefly discuss the algorithm for compiler design. Section IV presents the compiler construction tools. Section V gives overview of the Compiler Project at Leading Computer Science Universities. Finally, Section VI provides the conclusion of this paper.

2 Related Work

Yunsik Son *et al.* [4] provided a brief overview on symbol table. Symbol table is an essential module in compiler construction. It includes phases like lexical analysis, syntax analysis, semantic analysis and code generation. In this paper, they deal with reverse technique for the verification of the symbol table in objective C compiler.

They also discuss the design and implementation of a reverse translator that verifies and analyses the symbol table designed during the development stage objective C compiler. Furthermore, based on the symbol table verification, a correct code can be generated by examining the use of identifiers and attribute in the code generation step.

I. Budiselic *et al.* [5] discussed the experiences with the programming language instruction over the last three years and tool based assignment used before and quantitative differences in results. They also discuss the compiler design courses providing an overview on compiler project at seven different computer science universities in Europe and US. They also provide an overview on programming language translation courses, organization of PTL courses and describe the evolution of programming from its initial stage to its current design. Two important classifications of compiler design project and courses are explained. They roughly divide compiler design into two courses front end heavy and back end heavy.

Chengyong Wu *et al.* [6] presented an overview of the design of the main components of ORC, especially new features in the code generator. The Open Research Compiler (ORC) was jointly developed by the Intel Microprocessor Laboratory of Technology and the Institute of Computer Technical Academy of Science of China. It has become the leading open source compiler in processor family Itanium TM. ORC development methodology which is important for achieving the objectives is discussed. Performance comparison with other IPF compiler and a brief summary of research based on ORC are also presented.

John S. Mallozzi *et al.* [7] talked about one semester course in compiler design presents difficulties to an instructor who want to assign a project in which object oriented techniques are used. This paper describe a method that uses the tool developed by the author to generate a parser that encourages an object-oriented approach, clearly related code written by the student which automatically generate code with intended students to increase understanding.

Miodrag Djukic *et al.* [8] described a technique in which significance of controllability and speed is placed upward the retarget ability and cycle-accuracy to provide a better platform for software development. Many simulation instructional approaches place the retarget ability and cycle precision as the key functions to facilitate the exploration and performance of architecture and also estimate early in the development phase of hardware. The main idea of this work is to associate the simulator effort compiled with the development of the C and build target language compiler for the processor using knowledge related to compiler and reusing some common software elements.

Mirko Viroli [3] provided a brief description about EGO compiler (Extract Generic On-Demand). This is the result of a project developed in partnership with Sun Microsystems in order to evaluate a smooth support for generic time function, which does not require changes in the JVM or any other component of the Java Runtime Environment. We conceive and develop solution which is a sophisticated translation based on the type style step also known as reification of type parameters, where the type information is at runtime and automatically create as on per code and cached for future use. The main aspects of development are presented, from the basic design to implementation and deployment issues.

Johgheen Youn *et al.* [9] presented a new coding scheme and instructions based on the dynamic implied addressing mode (DIAM) to solve the limited space coding and side effects by trimming. Also introducing two versions of architectures to support our approach is based on DIAM. They also suggest a generation of code algorithm to fully utilize DIAM. In their work, architecture with DIAM exhibition shows code size reduction up to 8% and 18% on average speed compared with the basic architecture without DIAM.

Hankjin Lee *et al.* [10] provided a well established algorithm more over a methodology that is used for detection of design patter. In this paper, reclassification of GoF pattern takes place. Gang of Four (GoF) is known to be very useful for the detection of projects with reverse engineering methods. He also proposed GoF pattern detection technique. After that, evaluation of new technique is done and paper is concluded with the pros and cons of new approach, and what other work is to be done in terms of future research.

Ivan Keimek *et al.* [11] provided a brief description that reverse engineering is used in many fields of IT every day like binary code patching, legacy compatibility, network protocol analysis, malware analysis, rapid prototyping or in debugging. Despite its widespread use, reverse engineering is not actively taught as part of computer courses. This paper attempts to provide an overview of real life scenario of reverse engineering. Analysis of skills, ways of thinking that can be developed by reverse engineering and provides example that you can teach reverse engineering by resolution of practical problems. They also focus on the importance of reverse engineering as a tool to turn the self motivation in students and systematically build your logical thinking skills and analytical skills.

Cristina Cifuentes *et al.* [12] presented different type of reverse engineering based on level of code abstraction, which was used to reengineer assembly code, CASE code, machine code and source code. In this paper they elaborate various type of reverse engineering and protection for copyright software. Common uses of reverse engineering were explained. Comparative overview of the legal standing reverse engineering are provided. They also propose the existing and future challenges of the global electronic community for the protection of digital works.

3 Algorithm for Compiler Designing Process

Compiler design and related set of classic algorithms provides a pretty flexible software architecture that can be called “abstract machine”architecture. Sometimes using this architecture and adapting it to a particular task can make design more transparent and more easily debugged.

a. **Memory Aware Mapping for Compiler**

SOS: Set Operations to Schedule
QRO: Query Ready to Schedule Operations
G: Application DDG

Pseudo code for Memory Aware Mapping

```
Begin
Priorities Assign (G);
p=Highest Priority // Minimum mobility
while (SOS ≠ ϕ)
{
QRO = queue ROP (p)
  do
   {
    O_p = dequeue QRO
    (DDP_s , RTime) = Predecessors (O_p)
    (IDDP_s ) = Predecessors – R (O_p)
    do
     {
Choice=GetCost(DDP_S,IDDP_s,  RTime);
          RTime++
       }
while ( Resources_Cong (Choices));
              Dscn = DecidesScheduleTime (Choices)
              RsvResources (Decision)
              Sch(O_p)
              SOS = SOS - O_p
} while (QRO ≠ ϕ);
              p = p+1
}
End
```

The pseudo starts with DDG to represent the priority. DDG is initialized by setting the minimum value of mobility. QRO queue takes ROP function which has a value of mobility less than or equal to the value of variable p. After that do while loop schedule each operation once at a time until it become empty. DDPs set the earliest clock cycle at which operation Op can be schedule. The preprocessor returns Op where IDDPs were executed. Subsequently, GetCost return choice variables. After that RTime is incremented and the GetCost function is repeated until available Op are found. The decision schedule time function analyze the mapping costs from choice variable and select the most efficient operation.

b. **Distributed Shared Memory Automatic Parallelizing Compiler**

T_a : Target Array
P_O : Parallelizing Loop
K_O: Kernel Loop
DDM: Data Distributed Method
IK_O : Intraprocedural Kernel Loop
iK_o : Interprocedural Kernel Loop
FTC: First Touch Control

Pseudo code for Distributed Shared Memory Automatic Parallelizing Compiler

```
Begin
Detect $\longrightarrow P_o$ and $T_a$
For
        DDM=Determination of the shape
        Determination of the $IK_o$
        if
                $T_a \longrightarrow$ Arg
                return determination of $iK_O$
        else
                no change
        endif
end for DDM
Err $\longrightarrow$ check distribution
if
        Code $\longrightarrow$ FTC
        Analysis of data redistribution
else
        B $\longrightarrow$ Analysis of reference pattern
        Return B
end if
End
```

In this pseudo code, we start with the detection of parallelizing loops and target arrays. Parallelizing loop for each shape of data distribution is determined. Kernel loop is distributed and data redistribution analysis is done. In the end, analysis of first touch control for data redistribution and reference pattern.

c. Demand Driven to Detect Parallelism in Irregular Code of Compiler

Pseudo Code For Demand Driven to Detect Parallelism in Irregular Code

```
Begin
Case1:
Check SCC ($X_1$,…….,$X_n$)
when dependence is found
SCC($Y_1$,…………,$Y_m$) $\longrightarrow$ SCC($X_1$,………,$X_n$)
L $\longrightarrow$ start classification
        if
        SCC is already in stack
        Return
        SCC($X_1$,…….,$X_n$) SCC($Y_1$,……..,$Y_m$)
        else
        B $\longleftarrow$ Generate Error
        endif
Case2:
SCC($X_1$) $\longleftarrow$ trival component
        inherit $CLASS_{X1}$(e)
        $X_1$=e
      if
      SCC($X_1$,……..,$X_n$) $\longleftarrow$ non trival component
      A $CLASS_{XK}$ ($X_K(S_K)=e_K$)
      Return $\longleftarrow$ classified component
      else
      SCC $\longleftarrow$ return unclassified
      endif
End
```

In this pseudo code, non classified component is pushed into stack and process is started. If SCC is independent then its classification is found. Classification process starts when SCC (Y_1,………,Y_m). It reaches a deadlock when mutually dependent SCC exists in the loop. If dependence is found, stack is checked before starting classification. If already in stack, dependence exists. In case 2 if stack belong to the same class then component inherit otherwise remains unclassified.

4 Compiler Construction Tools

There are so many tools for compiler design; few of them are listed below [18] :

i. Lex & Yacc: Lex and Yacc are the most classic UNIX tools for the compiler construction. Lex does tokenization which helps to create programs whose control flow is handled by instances of regular expression in the input stream. Yacc provides a parsing tool to illustrate the input to a computer program. The Yac user specifies the grammar of the input with its code to be invoked as each structure in that grammar is renowned. Yacc provides specification into a subroutine to process the input.

ii. Lemon: The lemon program is an LALR parser generator. It takes a context free grammar and converts it into a subroutine that will parse a file using that grammar. Lemon is analogous to much more programs like “BISON” and “YACC”, but the lemon is not companionable with either bison or yacc.

iii. GCC-RTL: RTL store text in a file as an interface between the language front end and GNUCC. GNUCC was designed to use RTL internally only.

iv. ANTLR: ANother Tool for Language Recognition. It is a powerful parser generator for processing, reading, executing, translating binary files or structure files. It’s widely used to build tools, languages and frameworks.

v. ML-RISC: MLRISC is a customization optimization back-end written in Standard ML and has been successfully retargeted to multiple architectures like PPC, Sparc, Alpha, MIPS.

5 Overview of The Compiler Project at Leading Computer Science Universities

In this section, we briefly summarize the compiler design courses and accompanying programming assignments from some of the leading computer science universities in the US and Europe. The specifics of each course are shown in Table I.

Table 1. Key Properties of Compiler Courses in Some Computer Science Universities.

Universities	Course name	Source Language	Target Language	Implementation Language	Lecture Front end %	Tools	Student per group	Project Impact
MIT	6.035 Computer Language Engineering	Decaf	x86-64	Java	10	ANTLR	3–4	60
CMU	15-411 Compiler Design	L1-L4	X86-64	SML, Ocaml, Haskell, Java and others	10	Lexer and parser generators	1–2	70
Columbia	COMS W4115 Programming Languages and Translators	Student designed	Student Choice	Ocalm	25	ocamllex, ocamlyacc	5	40
ETH Zurich	Compiler Design I	JavaLi	X86 or similar	Java	30	JLex, CUP	2	66
Stanford	CS 143 Compilers	C++, Java	COOL	MIPS	30	Lexer and parser generators	1-2	50
Berkeley	CS 164 Programming Languages and Compilers	COOL	MIPS	Java	30	JLex, CUP	1-2	40
Oxford	Compiler	Oberon-like	Keiko/ARM	Ocalm	15	ocamllex, ocamlyacc	N/A	N/A

6 Conclusion

In this paper, we conclude that compiler is a program that translates a source-code written in programming-language (like C or Pascal) to an object-file. Afterwards a linker links the object-file with other object-files and libraries to make them executable (like COM or EXE). Many compilers will perform both the compiling and linking steps in one operation. A compiler must do much more checking about the legality of the statements, make calls to functions, import from libraries, and manage variables of different scopes and so on. Today, compiler design is a vast field of research. Every programming language need different compiler to run a program. So, we need generic compiler to make programmer's work easy. The current paper gives a prerequisite environment for the construction and design of an efficient generic compiler.

References

1. Takashi Hirooka *et al.* "Automatic Data Distribution Method Using First Touch Control For Distributed Shared Memory Multiprocessor", pp. 147–161, Springer 2003.
2. Manuel Arenaz *et al.* "A Compiler Framework to Detect Parallelism in Irregular Code", pp. 306–320, Springer 2003.
3. Mirko Viroli "Effective and efficient compilation of run-time generics in Java", Electronic Notes in Theoretical Computer Science, vol. 2, pp. 95–116, 2005.
4. Son, Yunsik, Seman Oh, and Yangsun Lee, "A Reversing Technique for Symbol Table Verification on Compiler Constructions" 2014.
5. Budiselic, I., D. Skvorc, and S. Srbljic "Designing the programming assignment for a university compiler design course", 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), IEEE, 2014.
6. Wu, Chengyong, *et al*. "An overview of the open research compiler", Languages and Compilers for High Performance Computing Springer Berlin Heidelberg, pp. 17–31, 2005.
7. Mallozzi, John S. "Thoughts on and tools for teaching compiler design", Journal of Computing Sciences in Colleges , vol. 21.2, pp. 177–184, Springer 2005
8. Djukic, Miodrag, *et al*. "An Approach to Instruction Set Compiled Simulator Development Based on a Target Processor C Compiler Back-End Design", First IEEE Eastern European Conference, IEEE, 2009.
9. Youn, Jonghee M., *et al*. "Two versions of architectures for dynamic implied addressing mode", Journal of Systems Architecture, vol. 8, pp. 368–383, 2010.Leupers, Rainer. "Compiler design issues for embedded processors", Design & Test of Computers, vol. 4, pp 51–58, IEEE, 2002.
10. Lee, Hakjin, Hyunsang Youn, and Seunghwa Lee, "Automatic detection of design pattern for reverse engineering", 5th ACIS International Conference on Software Engineering Research, Management & Applications, IEEE, 2007.
11. Ivan Klimek, Marián Keltika and František Jakab, "Reverse Engineering as an Education Tool in Computer Science", 9th IEEE International Conference on Emerging eLearning Technologies and Applications, pp. 123–126, IEEE2007.
12. Cifuentes, Cristina, and Anne Fitzgerald. "The legal status of reverse engineering of computer software", vol. 2, pp. 337–351, Springer, 2000.
13. Kim Ann Zimmerman June 04, 2012 ,"History of computing", Retrieval 21st May 2015, http://www.livescience.com/20718-computer-history.html.

14. Jeremy Meyers "A Brief History Of The Computers(B.C.–1993 A.D)", Retrieval 20 May 2015, http://www.jeremymeyers.com/comp.
15. Mary Bellis "History of Computers" Retrieval 15th May 2015, http://inventors.about.com/library/blcoindex.htm.
16. Scribd, Feb24,2012 "Computer History Table", Retrival 10th May 2015, http://www.scribd.com/doc/82649888/Computer-History-Table#scribd.
17. Vangie Beal "Compiler", Retrieval 22th May 2015, http://www.webopedia.com/TERM/C/compiler.html
18. Christopher B. Browne's "Compiler Construction Tools", Retrieval 22th May 2015, http://linuxfinances.info/info/compilingtools.html