COMPILERS-STUDY FROM ZERO

Vishal Sharma, Priyanka Yadav, Priti Yadav
Computer Science Department, Dronacharya College of Engineering/ Maharishi Dayanand University, India

***Abstract*- This paper gives the basic understanding of compilers. The beginning of paper explain the term compiler followed by its working, need and knowledge required in building compilers. Further we have explained the different types of compiler and the application of compilers. The paper shows the study of different types of compilers.Thus, going through this paper one will end up with a good understanding of compilers and their future aspect. .**

***Index Terms*- Analyser, Compilers, FORTRAN, LISP, UNIVAC.**

I. INTRODUCTION

Compilers are fundamental to modern computing. They act as *translators*, transforming human-oriented *programming languages* into computer-oriented *machine languages.*

A compiler allows programmers to ignore the machine-dependent details of programming. Compilers allow programs and programming skills to be *machine-independent*.

Compilers also aid in detecting programming errors (which are all too common).

Compiler techniques also help to improve computer security.

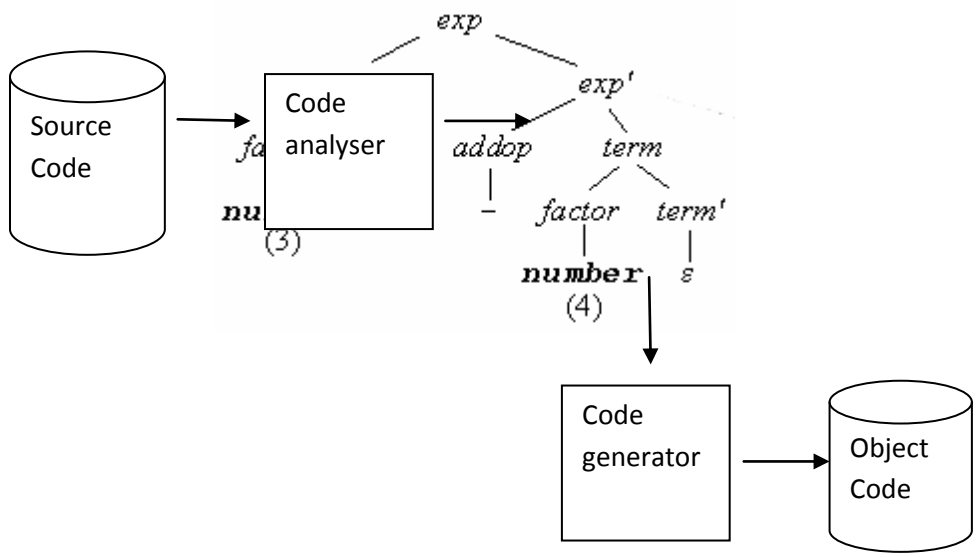
For example, the Java Byte code Verifier helps to guarantee that Java security rules are satisfied.

II. HOW DOES A COMPILER WORK?

A compiler can be viewed in two parts:

1. Source Code Analyser: which takes as input source code as a sequence of characters, and interprets it as a structure of symbols (vars, values, operators, etc.)

2. Object Code Generator: which takes the structural analysis from (1) and produces runnable code as output?



The Code Analyser typically has three stages:

• Lexical Analysis: accepts the source code as a sequence of chars, outputs the code as a sequence of tokens.

• Syntax Analyser: interprets the program tokens as a structured program.

• Semantic Analyser: checks that variables are instantiated before use, etc.

What is the Challenge?

Many variations:

- many programming languages (eg, FORTRAN, C++, Java)
- many programming paradigms (eg, object-oriented, functional, logic)
- many computer architectures (eg, MIPS, SPARC, Intel, alpha)
- many operating systems (eg, Linux, Solaris, Windows)

Qualities of a compiler (in order of importance):

1. the compiler itself must be bug-free
2. it must generate correct machine code
3. the generated machine code must run fast
4. the compiler itself must run fast (compilation time must be proportional to program size)
5. the compiler must be portable (ie, modular, supporting separate compilation)
6. it must print good diagnostics and error messages
7. the generated code must work well with existing debuggers
8. must have consistent and predictable optimization.

Building a compiler requires knowledge of

- programming languages (parameter passing, variable scoping, memory allocation, etc)
- theory (automata, context-free languages, etc)
- algorithms and data structures (hash tables, graph algorithms, dynamic programming, etc)
- computer architecture (assembly programming)
- software engineering.

A compiler can be viewed as a program that accepts a source code (such as a Java program) and generates machine code for some computer architecture. Suppose that you want to build compilers for *n* programming languages (eg, FORTRAN, C, C++, Java, BASIC, etc) and you want these compilers to run on *m* different architectures (eg, MIPS, SPARC, Intel, alpha, etc). If you do that naively, you need to write *n***m* compilers, one for each language-architecture combination.

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:

1. *scanning*: a scanner groups input characters into tokens;
2. *parsing*: a parser recognizes sequences of tokens according to some grammar and generates *Abstract Syntax Trees* (ASTs);
3. *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;
4. *optimization*:
5. Optimizes IRs.

The back end consists of the following phases:

1. *instruction selection*: maps IRs into assembly code;
2. *code optimization*: optimizes the assembly code using control-flow and data-flow analyses, register allocation, etc;
3. *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 sytem 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.

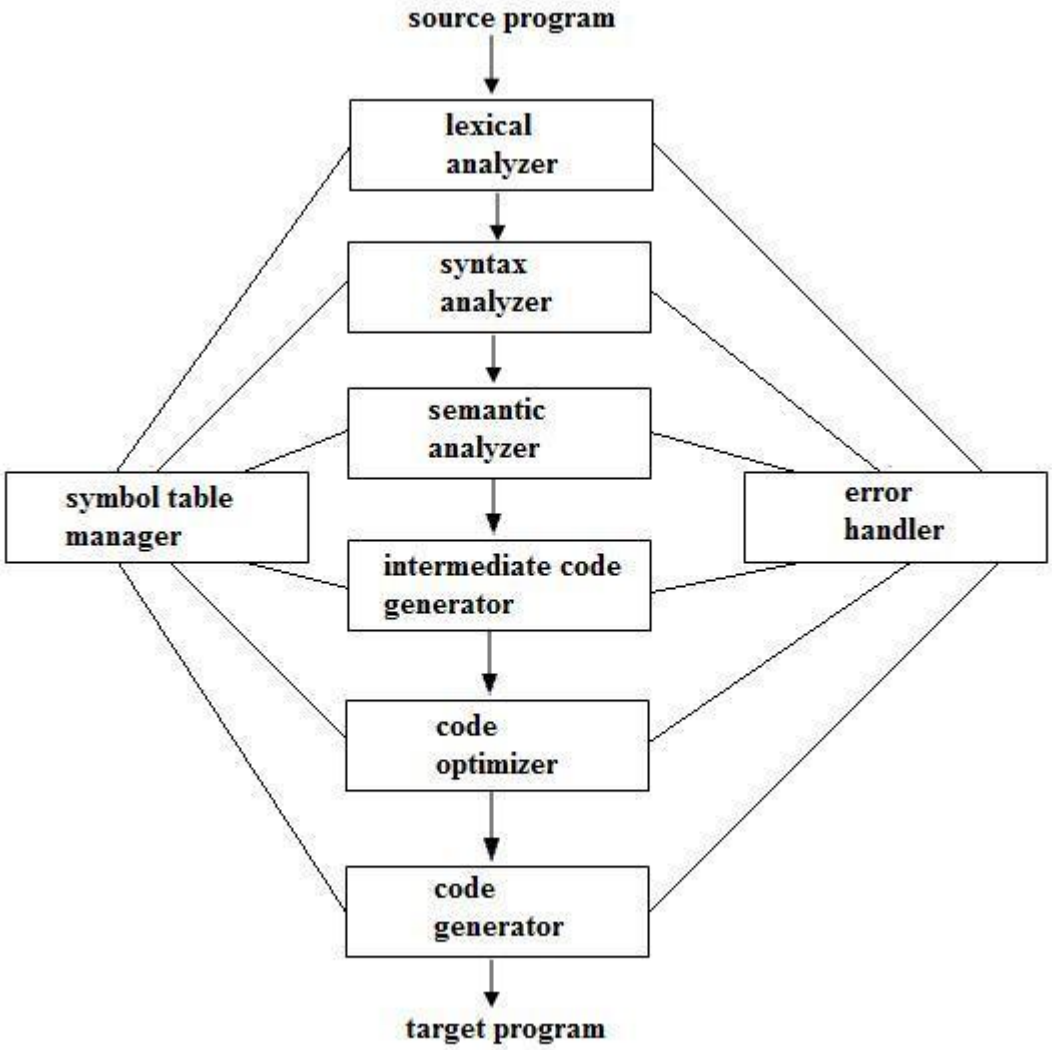


Fig 1.5 Phases of a compiler

In computer science, **bootstrapping** is the process of writing a compiler (or assembler) in the target programming language which it is intended to compile. Applying this technique leads to a self-hosting compiler.

Many compilers for many programming languages are bootstrapped, including compilers for BASIC, ALGOL, C, Pascal, PL/I, Factor, Hask ell, Modula-2, Oberon, OCaml, Common Lisp, Scheme, Java, Python, Scala, Nimrod and more.

NELIAC

The Navy Electronics Laboratory International ALGOL Compiler or NELIAC was a dialect and compiler implementation of the ALGOL 58 programming language developed by the Naval Electronics Laboratory in 1958.

NELIAC was the brainchild of Harry Huskey — then Chairman of the ACM and a well known computer scientist, and supported by Maury Halstead, the head of the computational centre at NEL. The earliest version was implemented on the prototype USQ-17 computer (called the Countess) at the laboratory. It was the world's first self-compiling compiler. This means that the compiler was first coded in simplified form in assembly language (the bootstrap), and then re-written in its own language, compiled by this bootstrap compiler, and re-compiled by itself, making the bootstrap obsolete.

Lisp

The first self-hosting compiler (excluding assemblers) was written for Lisp by Tim Hart and Mike Levin at MIT in 1962.[4] They wrote a Lisp compiler in Lisp, testing it inside an existing Lisp interpreter. Once they had improved the compiler to the point where it could compile its own source code, it was self-hosting.

The compiler as it exists on the standard compiler tape is a machine language program that was obtained by having the S-expression definition of the compiler work on itself through the interpreter. (AI Memo 39)

This technique is only possible when an interpreter already exists for the very same language that is to be compiled. It borrows directly from the notion of running a program on itself as input, which is also used in various proofs in theoretical computer science, such as the proof that the halting problem is undecidable.

XPL

XPL is a dialect of the PL/I programming language, developed in 1967, used for the development of compilers for computer languages. It was designed and implemented by a team with William McKeeman, James J. Horning, and David B. Wortman at Stanford University and the University of California, Santa Cruz. It was first announced at the 1968 Fall Joint Computer Conference in San Francisco.

It is the name of both the programming language and the LALR parser generator system (or TWS: translator writing system) based on the language.

XPL featured a relatively simple bottom-up compiler system dubbed MSP (mixed strategy precedence) by its authors. It was bootstrapped through Burroughs Algol onto the IBM System/360 computer. Subsequent implementations of XPL featured an SLR(1) parser.

XCOM

The XPL compiler, called XCOM, is a one-pass compiler using a table-driven parser and simple code generation techniques. Versions of XCOM exist for different machine architectures, using different hand-written code generation modules for those targets. The original target was IBM System/360.

XCOM compiles from XPL source code, but since XCOM itself is written in XPL it can compile itself – it is a self-compiling compiler, not reliant on anyone else's compilers. Several famous languages have self-compiling compilers, including Burroughs B5000 Algol, PL/I, C, LISP, and Java. Creating such compilers is a chicken-and-egg conundrum. The language is first implemented by a temporary compiler written in some other language, or even by an interpreter (often an interpreter for an intermediate code, as BCPL can do with intcode or O-code). XCOM began as an Algol program running on Burroughs machines, translating XPL source code into System/360 machine code. Someone manually turned its Algol source code into XPL source code. That XPL version of XCOM was then compiled on Burroughs, creating a self-compiling XCOM for System/360 machines. The Algol version was then thrown away, and all further improvements happened in the XPL version only. This is called bootstrapping the compiler. Retargeting the compiler for a new machine architecture is a similar exercise, except only the code generation modules need to be changed.

XCOM is a one-pass compiler (but with an emitted code fix-up process for forward branches, loops and other defined situations). It emits machine code for each statement as each grammar rule within a statement is recognized, rather than waiting until it has parsed the entire procedure or entire program. There are no parse trees or other required intermediate program forms, and no loop-wide or procedure-wide optimizations. XCOM does, however, perform peephole optimization. The code generation response to each grammar rule is attached to that rule. This immediate approach can result in inefficient code and inefficient use of machine registers. Such are offset by the efficiency of implementation, namely, the use of dynamic strings mentioned earlier: in processing text during compilation, substring operations are frequently performed. These are as fast as an assignment to an integer; the actual substring is not moved. In short, it is quick, easy to teach in a short course, fits into modest-sized memories, and is easy to change for different languages or different target machines.

Who developed the first language compiler and when?

Rear Admiral **Dr. Grace Murray Hopper**, in 1949. The compiler, written in assembly language,

converted symbolic mathematical code into machine code.

By 1949 programs contained mnemonics that were transformed into binary code instructions executable by the computer. Admiral Hopper and her team extended this improvement on binary code with the development of her first compiler, the A-O. The A-O series of compilers translated symbolic mathematical code into machine code, and allowed the specification of call numbers assigned to the collected programming routines stored on magnetic tape. One could then simply specify the call numbers of the desired routines and the computer would "find them on the tape, bring them over and do the additions. This was the first compiler," she declared.

III. DIFFERENT TYPES OF COMPILERS

One-pass compiler

A one-pass compiler (also known as "narrow compiler") is a compiler that passes through the source code of each compilation unit only once. Due to this reason it is very fast. But the down-side of this is that mostly, it can only be used in simple programs.

Threaded code compiler

This type of compiler replaces given strings in the source with given binary code. Many FORTH implementations use threaded code and some use the term threading for almost any technique used to implement Forth's virtual machine.

Incremental compiler

This type of compiler was used in Lisp systems. It continued to be developed for imperative and interactive programming.

Stage compiler

This is used in Prolog machines for instance. The process involves compiling to assembly language.

Just-in-time compiler

Starting off in bytecode, and then compiling it into machine code just in time before the execution starts off.

Retargetable compiler

This type of compiler is used to produce code from various CPUs. However it is worth noting that the object code in this case is sometimes of a lower quality. This is due to the fact that this type of compiler is quite generic, as opposed to a compiler that was designed specifically for a particular processor.

Parallelizing compiler

This kind of compiler is used in systems composed of what is known as a parallel architecture. The serial input program is converted in such a way that it can be processed efficiently by the system.

There are several types of compilers, each having their particular characteristics to suit varying needs and systems. An overview can be found in the Cambridge encyclopaedia volumes and other sites, which describe their structure, techniques, possible errors and uses.

IV. APPLICATIONS OF COMPILER TECHNOLOGY

1: Implementation of High-Level Programming Languages

- Abstraction: All modern languages support abstraction. Data-flow analysis permits optimizations that significantly reduce the execution time cost of abstractions.
- Inheritance: The increasing use of smaller, but more numerous, methods has made interprocedural analysis important. Also optimizations have improved virtual method dispatch.
- Array bounds checking in Java and Ada: Optimizations have been produced that eliminate many checks.
- Garbage collection in Java: Improved algorithms.
- Dynamic compilation in Java: Optimizations to predict/determine parts of the program that will be heavily executed and thus should be the first/only parts dynamically compiled into native code.

2: Optimization for Computer Architectures

Parallelism

Major research efforts had lead to improvements in

- Automatic parallelization: Examine serial programs to determine and expose potential parallelism.
- Compilation of explicitly parallel languages.

Memory Hierarchies

All machines have a limited number of registers, which can be accessed much faster than central memory. All but the simplest compilers devote effort to using this scarce resource effectively. Modern processors have several levels of caches and advanced compilers produce code designed to utilize the caches well.

3: Design of New Computer Architectures

RISC (Reduced Instruction Set Computer)

RISC computers have comparatively simple instructions, complicated instructions require several RISC instructions. A CISC, Complex Instruction Set Computer, contains both complex and simple instructions. A sequence of CISC instructions would be a larger sequence of RISC instructions. Advanced optimizations are able to find commonality in this larger sequence and lower the total number of instructions. The CISC Intel x86 processor line 8086/80286/80386/... had a major change with the 686 (a.k.a. pentium pro). In this processor, the CISC instructions were decomposed into RISC instructions *by the processor itself*. Currently, code for x86 processors normally achieves highest performance when the (optimizing) compiler emits primarily simple instructions.

Specialized Architectures

A great variety has emerged. Compilers are produced *before* the processors are fabricated. Indeed, compilation plus simulated execution of the generated machine code is used to evaluate proposed designs.

4: Program Translations

Binary Translation

This means translating from one machine language to another. Companies changing processors sometimes use binary translation to execute legacy code on new machines. Apple did this when converting from Motorola CISC processors to the PowerPC. An alternative is to have the new processor execute programs in both the new and old instruction set. Intel had the Itanium processor also execute x86 code. Apple, however, did not produce their own processors.

With the recent dominance of x86 processors, binary translators from x86 have been developed so that other microprocessors can be used to execute x86 software.

Hardware Synthesis

In the old days integrated circuits were designed by hand. For example, the NYU Ultracomputer research group in the 1980s designed a VLSI chip for rapid interprocessor coordination. The design software we used essentially let you *paint*. You painted blue lines where you wanted metal, green for polysilicon, etc. Where certain colors crossed, a transistor appeared.

Current microprocessors are much too complicated to permit such a low-level approach. Instead, designers write in a high level description language which is compiled down the specific layout.

Database Query Interpreters

The optimization of database queries and transactions is quite a serious subject.

Compiled Simulation

5: Software Productivity Tools

Dataflow techniques developed for optimizing code are also useful for finding errors. Here correctness is not an absolute requirement, a good thing since finding all errors in undecidable.

Type Checking

Techniques developed to check for type correctness (we will see some of these) can be extended to find other errors such as using an uninitialized variable.

Bounds Checking

As mentioned above optimizations have been developed to eliminate unnecessary bounds checking for languages like Ada and Java that perform the checks automatically. Similar techniques can help find potential buffer overflow errors that can be a serious security threat.

Memory-Management Tools

Languages (e.g., Java) with garbage collection cannot have memory leaks (failure to free no longer accessible memory). Compilation techniques can help to find these leaks in languages like C that do not have garbage collection.

V. Study of Different Compilers and Their Properties:

Common Lisp Compiler

Common Lisp (CL) is a dialect of the Lisp programming language, published in ANSI standard document "ANSI INCITS 226-1994 (R2004), (formerly X3.226-1994 (R1999))".[1] From the ANSI Common Lisp standard the Common Lisp HyperSpec has been derived[2] for use with web browsers. Common Lisp was developed to standardize the divergent variants of Lisp (though mainly the MacLisp variants) which predated it, thus it is not an implementation but rather a language specification. Several implementations of the Common Lisp standard are

available, including free and open source software and proprietary products.

Common Lisp is a general-purpose, multi-paradigm programming language. It supports a combination of procedural, functional, and object-oriented programming paradigms. As a dynamic programming language, it facilitates evolutionary and incremental software development, with iterative compilation into efficient run-time programs.

Compiler	Author	Target	Windows	Unix-like	Other OSs	License type	IDE?
Allegro Common Lisp	Franz, Inc.	Native code	Yes	Yes	Yes	Proprietary	Yes
Armed Bear Common Lisp		JVM	Yes	Yes	Yes	GPL	Yes
CLISP		Bytecode	Yes	Yes	Yes	GPL	No
Clozure CL	Clozure Associates	Native code	Yes	Yes	No	LGPL	Yes
CMU Common Lisp		Native code, Bytecode	No	Yes	No	Public Domain	Yes
Corman Common Lisp		Native code	Yes	No	No	Proprietary	Yes
Embeddable Common Lisp		Bytecode, C	Yes	Yes	Yes	LGPL	Yes
GNU Common Lisp		C	Yes	Yes	No	GPL	No
LispWorks	LispWorks Ltd	Native code	Yes	Yes	No	Proprietary	Yes
Open Genera	Symbolics	Ivory emulator	No	Yes	No	Proprietary	Yes
Scieneer Common Lisp	Scieneer Pty Ltd	Native code	No	Yes	No	Proprietary	No
Steel Bank Common Lisp		Native code	Yes	Yes	Yes	Public Domain	Yes

D- COMPILERS

The D programming language is an object-oriented, imperative, multi-paradigm system programming language created by Walter Bright of Digital Mars. Though it originated as a re-engineering of C++, D is a distinct language, having redesigned some core C++ features while also taking inspiration from other languages, notably Java, Python, Ruby, C#, and Eiffel.

D's design goals attempt to combine the performance of compiled languages with the safety and expressive power of modern dynamic languages. Idiomatic D code is commonly as fast as equivalent C++ code, while being shorter and memory-safe.

Compiler	Author	Windows	Unix-like	Other OSs	License type	IDE?
Digital Mars D	Digital Mars and	Yes	32-bit Linux, Mac	No	GPL and Artistic	No

Compiler	Author	Windows	Unix-like	Other OSs	License type	IDE?
(DMD)	others		OS X, FreeBSD			
D Compiler for .Net	?	Yes	Yes	?	?	?
GDC	GCC	Yes	Yes	No	GPL	No
LDC	LLVM	Yes	Yes	No	Open Source	No

DiBOL – COMPILERS

DiBOL or Digital's Business Oriented Language is a general-purpose, procedural, imperative programming language, which is well-suited for Management Information Systems (MIS) software development. It has a syntax similar to FORTRAN and BASIC, along with BCD arithmetic. It shares the COBOL program structure of data and procedure divisions.

Compiler	Author	Windows	Unix-like	Other OSs	License type	IDE?
Synergy DBL[2][3][4]	Synergex	Yes	Yes	Yes	Proprietary	Yes

EIFFEL

Eiffel is an ISO-standardized, object-oriented programming language designed by Bertrand Meyer (an object-orientation proponent and author ofObject-Oriented Software Construction) and Eiffel Software. The design of the language is closely connected with the Eiffel programming method. Both are based on a set of principles, including design by contract, command-query separation, the uniform-access principle, the single-choice principle, the open-closed principle, and option-operand separation.
Many concepts initially introduced by Eiffel later found their way into Java, C#, and other languages. New language design ideas, particularly through the Ecma/ISO standardization process, continue to be incorporated into the Eiffel language.

Compiler	Author	Windows	Unix-like	Other OSs	License type	IDE?
EiffelStudio	Eiffel Software / Community developed (sourceforge)	Yes	Yes	Yes	Dual GPL / Proprietary	Yes

Java compilers

Compiler	Author	Windows	Unix-like	Other OSs	License type	IDE?
GNU Java	GNU Project	No	Yes	No	GPL	No
Javac	Sun Microsystems (Owned by Oracle)	Yes	Yes	Yes	GPL	No
S.N Java Compiler	SN Ink. (Owned by S.N)	Yes	No	No	Free	No
ECJ (Eclipse Compiler for Java)	Eclipse project	Yes	Yes	Yes	EPL	Yes

Pascal compilers

Compiler	Author	Windows	Unix-like	Other OSs	License type	IDE?
Amsterdam Compiler Kit	Andrew Tanenbaum and Ceriel Jacobs	No	Yes	Yes	BSD	No
Embarcadero Delphi	Embarcadero (CodeGear)	Yes	No	?	Proprietary	Yes
Delphi Prism	RemObjects	Yes	Yes	Yes	Proprietary	Yes
FrameworkPascal	Framework Computers, Inc.	Yes	No	Yes (MS-DOS)	Proprietary	Yes
Free Pascal	Florian Paul Klämpfl	Yes	Yes	Yes (OS/2, FreeBSD, Solaris, Haiku, etc.)	GPL	FPIDE & Lazarus
Irie Pascal	Irie Tools Limited	Yes	Yes	No	Proprietary	Yes
GNU Pascal	GNU Project	Yes	Yes	Yes	GPL	No
Kylix	Borland (CodeGear)	No	Yes (Linux)	No	Proprietary	Yes
Turbo Pascal for Windows	Borland (CodeGear)	Yes (3.x)	No	No	Proprietary	Yes
Microsoft Pascal	Microsoft	No	No	Yes (MS-DOS)	Proprietary	Yes
Neuron Pascal Compiler	Salah IBN AMAR	Yes	Yes	Yes	GPL	Yes
HP Pascal	Hewlett-Packard	No	No	Yes (OpenVMS)	Proprietary	Unknown
Turbo Pascal	CodeGear (Borland)	No	No	Yes	Freeware	Yes
Vector Pascal	Glasgow University	Yes	Yes	No	OpenSource	No
Virtual Pascal	Vitaly Miryanov and Allan Mertner	Yes	Yes (Linux)	Yes (OS/2)	Freeware	Yes
WDSibyl	Wolfgang Draxler and Speed-Soft	Yes	No	Yes (OS/2)	GPL	Yes

VI. CONCLUSION

Thus, we get an introduction about the compilers and their working. The modern world is changing very fast and hence this change is also taking place in the development of compilers. From the very first COMPILER for UNIVAC and FORTRAN to various latest compilers for each and every computer language, there has been significant growth in them. The compiler field must develop the technologies that enable more of the progress the field has experienced over the past 50 years.

Computer science educators must attract some of the brightest students to the compiler field by showing them its deep intellectual foundations, highlighting the broad applicability of compiler technology to many areas of computer science.

Today, compilers are fast, accurate and more precise to their working. With the upcoming technology the compiler growth will enhance and we can see the future of compilers as bright era.

REFERENCES

[1] Compiler Research: The Next 50 Years BY MARY HALL, DAVID PADUA, AND KESHAV PINGALI

[2] Compilers Spring term Mick O'Donnell: michael.odonnell@uam.es Alfonso Ortega: alfonso.ortega@uam.es

[3] http://www.cs.yale.edu/homes/tap/Files/hopper-story.html

[4] http://en.wikipedia.org/wiki/D_(programming_language)

[5] http://en.wikipedia.org/wiki/Common_Lisp

[6] http://en.wikipedia.org/wiki/History_of_compiler_construction

[7] http://en.wikipedia.org/wiki/List_of_compilers#Source-to-source_compilers