CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

Department of Information Engineering

Diploma Thesis:

PROGRAMMING LANGUAGE DESIGN AND COMPILER IMPLEMENTATION (AMHARIC BASED)

Author: Tofik Jemal AHMED

Supervisor: Pergl Robert, Dr. Ing.

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Department of Information Engineering

Faculty of Economics and Management

DIPLOMA THESIS ASSIGNMENT

Ahmed Tofik Jemal

Informatics

Thesis title

Programming Language Design and Compiler Implementation (Amharic Language Based)

Objectives of thesis

Design a programming language suitable for teaching computer science to Amharic speaking students and for practical tasks solving. Implement a lexical analyser, syntactic parser and intermediate code generator.

Methodology

Study the appropriate literature about the formal languages design and compilers implementation. Design a grammar of the language. Choose the appropriate implementation platform. Make the implementation and demonstrate it on a case-study.

Schedule for processing

Theoretic studies: Jan 2012 - Jul 2012
Grammar design: Sep 2012 - Oct 2012
Lexical analyser implementation: Nov 2012 - Dec 2012
Parser and compiler implementation: Jan 2013 - Mar 2013
Case study example: Mar 2013- Apr 2013

The proposed extent of the thesis
30

Keywords
compiler, programming language, lexical analysis, syntactic analysis

Recommended information sources

Mak R.: Writing Compilers and Interpreters, Wiley; 2 edition (August 10, 1996)
Appel A.W.: Modern Compiler Implementation in Java, Cambridge University Press; 2 edition (November 2002)
Vanicek et al.: Mathematical Foundations of Computer Science, Alfa Publishing, 2008

The Diploma Thesis Supervisor
Pergl Robert, Dr. Ing.

Last date for the submission
March 2013

Ing. Martin Pelikán, Ph.D.
Head of the Department

prof. Ing. Jan Hron, DrSc., dr.h.c.
Dean

Prague March 14, 2013

Declaration

I declare that I have worked on my diploma thesis titled “Programming Language Design and Compiler Implementation (Amharic Based)” by myself and I have only used the sources mentioned as list of references at the end of the thesis.

In Prague, ……..

……

Tofik Jemal AHMED

Acknowledgment

I would take the chance to thank several individuals who in one or another way contributed and extended their valuable assistance in the preparation and completion of this study.

First and foremost I would like to thank my supervisor, Dr. Ing. Robert Pergl, for his kind advice and supervision of my thesis, without his comment and appreciation this work would not have be possible.

PhDr. Vlastimil Černý, CSc. Head of International relations office, for his time and consultations for any single issues I brought to his office.

Ing. Martin Kozák and all members of the International relations office they have done their best to help me in my academic and social issues from the start to the end of my study.

It will be rude of me if I fail to thank for the man who provided the Czech translation of the summary. Ing. Gera Abate, thank you for the effort you have put to translate the summary.

My dad, Jemal Ahmed, my mom, Zubeyda Mohammed, my Sister, Sada Jemal, and my fiancé, Nurayni, it is nice of you to give me the freedom and time to concentrate on my work when you even most demand me. I guess it is with the prayers of yours I am blessed on happiness and success.

Last but not the least, my close friends, relatives and the one above all of us, the omnipresent Allah, for answering my prayers for giving me the strength to plod on despite my constitution wanting to give up and throw in the towel, thank you so much Dear Allah.

Programming Language design and Compiler Implementation (Amharic Based)

Návrh programovacího jazyka a implementace překladače (založeného na jazyku Amharic)

Summary

Amharic is an official language of The Federal Democratic Republic of Ethiopia (FDRE) and spoken in Ethiopia, Eritrea and some 2.7 million emigrants all over the world. It is the second most-spoken Semantic language next to Arabic. It uses Amharic Fidel for writing, which grew out of the Ethiopian Orthodox Church writing system called Ge'ez. The alphabets, fidels, of the languages are supported in Unicode from Code range 1200-137F Ethiopic and 1380-139F Ethiopic Supplement including the punctuations of the language. This thesis design a programming language that will use the Unicode character set of Ethiopic and Ethiopic Supplement and implement lexical scanner, syntactic parser and intermediate code generator.

The first part of the thesis will introduce Amharic language and the thesis in depth. The second part will explain the objective of the thesis and the methodology of the thesis. The next part is review of literature related to programming language design, compiler implementation and programming languages which are based on Unicode characters rather than ASCII code. Then it will explain and present the solution; BNF based grammar, the Lexer, Parser and code generator. Lastly it will draw conclusion and recommendations.

Keywords

Compiler, Amharic Based Programming language, Programming Language Design, Lexical analysis, Syntactic analysis, Parser, Unicode, BNF Grammar for Jempas, C, Jempas

Souhrn

Amharština je oficiálním jazykem Etiopské federativní demokratické republiky (FDRE) a mluvené v Etiopii, Eritrei a používá to přibližně 2,7 milionů emigrantů po celém světě. Po arabštině je to nejvíce používaný sémantický jazyk. Pro psání se používá Amharský Fidel (ABCDA), který pochází z Etiopské ortodoxní církevní systém psáni a jmenuje se Geéz. Abeceda je podporována unicodem v rozsahu 1200-137F Etiopské a 1380-139F doplňkové Etiopské abecedy včetně přesnosti jazyka. Tato diplomová práce vytvoří programovací jazyk, který používá Unicode znakové sady Etiopské a doplňkové Etiopské, implementuje lexikální scanner, syntaktický analyzátor a provádí střední generátor kódů.

První část práce zavede amharský jazyk a práce do hloubky. Druhá část vysvětlí cíl a metodiku práce. Další části obsahuje přehled literatury vztahující se k vývoji programovacího jazyka, implementace překladače a programovacích jazyků, které jsou založeny na Unicode znaky než ASCII kód. Pak vysvětli a presentuje řešení; gramatiky na základě BNF, scanner, analyzátor a generátor kódu. Konečně bude čerpat závěr a nabízí doporučení.

Klíčová slova

Překladač, Amharic založený programovací jazyk, programovací jazyk Design, lexikální analýza, syntaktická analýza, analyzátor, Unicode, BNF gramatiky, Jempas, C, Jempas

Table of Contents

Table of Figures

1. Introduction

Programming languages are how people talk to computers. The computer would be just as happy speaking any language that is unambiguous. One of the reason we have high level languages is because people can't deal with machine language. The point of programming languages is to prevent human brains from being overwhelmed by a mass of details. It will be great if we have a programming language that is closer to our mother tongue. Then programming will be easy to learn and fun to work with that is why designing a programming language and designing a compiler to it for Amharic speaking student is considered.

Amharic is a language spoken in Ethiopia and Eritrea. It is the second most spoken Semitic language next to Arabic with over 80 millions speakers. It uses a characters in Unicode character sets in range of 1200-137F as main set of characters and extended ranges are available time to time to support the symbols and dialects in the language. I am a native speaker of the language and will design a grammar for a Pascal like programming language in Amharic. The main objective is to design a programming language suitable to teach programming for Amharic speaking students and to implement its compiler in beta version. So at the end of the study there will be a language grammar in BNF to describe the vocabulary and syntax of the intended language, a compiler frontend which could translate the given language program in to three address code (TAC), sample test program and sample output in three address code.

Languages like Arabic, Chinese, Korean, Japanese and Amharic are totally based on Unicode and it is difficult for student from those cultures to learn the ASCII based programming language. Of course, there are a number of reasons for that; one of them is closed political system to foreign culture in the case of the Arab world, China and North Korea. The other reason could be lose of motivation towards a product based on others language, so developing a programming language that will be based on the students' mother tongue will raise the motivation by giving ownership. And difficulty to key in the characters was the main bottleneck but the current technological advancement had already

solved this problem for the owner of the language. However, it is still left a problem for non native.

The reader of this document could be anyone with interest of language theory, programming language design, language implementation, localization, internationalization but those with understanding of Amharic or any Unicode based language may benefit the most. One of the benefits is inspiration, since this research is only academic version, student in undergraduate and graduate level will be inspired to develop and they can use this as a startup. The other is Universities will use this as a supplementary material in teaching formal language theory, compiler construction and programming language courses. And it will enable researcher and software companies to see the market and research opportunity in Unicode based languages and its user like Ethiopian. The next benefit of this research and its output is that it will motivate and encourage students of the field to learn programming and programming languages which in turn will invite different stakeholders to invest in the fields of information technology. The last but not the least it will also have socio-economic advantage in advertising the available opportunity, market segment and potential skilled manpower in Unicode based language speaking country in this specific case Ethiopia, for localized products and for localization.

To my best knowledge this is the only in depth analysis made concerning programming language based in Amharic. And it is the first of its kind to produce expandable teaching supplement tool as its output. As compiler construction needs depth understanding of programming language, grammar, automata and characters, this thesis have contained a vast amount of information in literature review chapter. Most of the resource used there, are well recognized by the community of their area or have contributed to the development of the area in one or another way. Hence, there is no doubt on the authenticity of the information presented.

The rest of the document is organized in the following order. Objective and methodology is where the objective is properly set and the methodology, tools and method used are explained. Literature review is the part where different source of information are visited and presented to support the understanding of the upcoming parts. A chapter is dedicated for the work of the author, which describe and correlate each development with the presented literature. To understand this part one might need a basic understanding of compilation and programming but not limited. As it is designed non programmer in mind the language is simple, teaser and readable. There is a chapter that discuss on the result based on the practical work. So it will add in depth analysis of the work done by the author. The good practice will be encouraged and the bad path will be commented. The last chapter will present conclusion and recommendation. Which will extend the comments given in results chapter and will point out further research area in the field which could be the extension of this thesis, a branch or a stand alone. Reference list and appendixes are attached for further study on the concept used in this thesis. And the appendix might consist of a full front end of a compiler for the designed language, sample test program and output.

2. Objectives and Methodology

2.1. Objectives

The objective is to develop a programming language that can be used to teach programming, compiler construction and formal language theory for Amharic speaking students and composed of the grammar of the language in Backus-Naur form (BNF), the lexical analyzer, the syntax analyzer and the intermediate code generator.

2.2. Methodology

The main sources of information for the thesis are from literature reviews and Internet resources. The output is the development of a compiler with the capability of lexical analyzer; scanning wide characters from the given input file and converting them in to a sequence of tokens, syntax analyzer or parser; analyzing the syntax based on the given grammar rule and code generator, generating three address codes (TAC).

Different tools are used, to list them: c language; to write the lexer, parser, symbol table and code generating capability, BNF; to represent the grammar of the language, Kate text editor; as code editor; Terminal, to enter command and GCC compiler.

The first step during the development was to configure the development environment and make ready to support for Amharic input method. During the configuration stage installation of Ubuntu, VMware, Kate, intelligent input bus (ibus) and activation of the Amharic input method (Am) is performed.

The language, Jempas, is developed and tested in ubuntu 9.0 using VMware and should compile and run in any c compiler in Linux and Unix. The source code editor must be utf-8 compatible and the input method has to be set to Am or the Nyala font should be supported by the editor to write the source code properly. The source code file could have any extension but preferable to use .j to point that this language is Jempas.

Programming should be as easy as writing email in the next decades, that is why programming now becomes relatively simper from the age of Ada. So I have followed a modular approach in writing a program to make a program simple, readable and reusable. Due to the influence of knowledge in C++, Jempas is developed and organized in modules by using header files for public functions and variables.

Incremental development and test methodology is applied during the development of Jempas. Each part of the compiler is developed and tested separately then integrated and tested.

3. Literature Review

The study of programming language design and compiler writing touches upon linguistics, programming language, machine architecture, language theory, algorithms and software engineering. Those could be categorized broadly in theoretical computing, which consists of language theory, machine architecture and algorithms, and practical computing or software engineering. So the main focus of this thesis will be on theoretical computing.

3.1. Language

Noam Chomsky considered a language to be a set (finite or infinite) of sentences, each finite in length and constructed out of a finite set of elements. All natural languages in their spoken or written form are languages in this sense, since each natural language has a finite number of phonemes (or letters in its alphabet) and each sentence is representable as a finite sequence of these phonemes (or letters), though there are infinitely many sentences. Similarly, the set of sentences' of some formalized system of mathematics can be considered a language. The fundamental aim in the linguistic analysis of a language L is to separate the grammatical sequences which arc the sentences of L from the ungrammatical sequences which are not sentences of L and to study the structure of the grammatical sequences. The grammar of L will thus be a device that generates all of the grammatical sequences of L and none of the ungrammatical ones. (Chomsky, 1957)

3.2.Formal languages and grammars

The theory of formal languages originated in the study of natural languages. “The description of natural language is traditionally called a GRAMMAR; it should indicate how the sentences of a language are composed of elements, how elements from larger units, and how these units are related within the context of the sentence” (levelt, 2008). The theory of formal languages rose from the need to provide a formal mathematical basis for such description.

William, (Levelt, 2008), mentioned that Chomsky, the founder of the theory, was primarily concerned with a more thorough examination of the basis of linguistic theory. This involves such questions as "what are the goals of linguistic theory?", "what conditions must a grammar fulfill in order to be adequate in view of these goals?" and "what is the general form of a linguistic theory?"And said "without a formal basis, these and similar questions cannot be handled with sufficient precision."

A formal language can be used as a mathematical model for a natural language; while a formal grammar can act as a model for a linguistic theory. In computer science it is used as basis for defining programming languages and other systems in which the words of the language are associated with particular semantics.

From mathematical point of views grammars are formal systems like Turing machines, computer programs, prepositional logic, theories of inference and neural nets. A formal system is, broadly defined as any well-defined system of abstract thought based on the model of mathematics. "Formal systems characteristically transform a certain input into output by means of completely explicit, mechanically applicable rules" (levelt, 2008). Based on their input William, (levelt, 2008), had discussed three types of formal systems: Generative system, the input is an abstract start symbol and its output is a string of 'words' which constitutes a 'sentence' of the formal 'language'; Automata, use the sentences of a language as input and gives an abstract stop symbol as its output; and Grammatical Inference procedure, takes a sample of the sentences of a language as input and its output is a grammar which is in some way adequate for the language[1].

[1] The quotation marks around 'word', 'sentence', and 'language' indicate that these terms are not used in their full linguistic sense, but rather are concepts which must be strictly defined within the formal system (Levelt, 2008).

A formal definition of a grammar and its building blocks are given in Chomsky (1956, 1957) as a grammar $G = (V_N , V_T , P, S)$ is a system consisting of a nonterminal vocabulary V_N , a terminal vocabulary V_T , a set of productions P, and a start symbol S, with the following properties:

1. V_N, V_T and P are finite, nonempty sets.

2. $V_N \cap V_T = 0$.

3. $P \subset V^+ \times V^*$.

4. $S \in V_N$.

A sentence generated by G is every element s of V*T for which $S \ast\Rightarrow s$, i.e. it is a terminal string derivable from S by the productions of P. The language L(G) generated by G is the set of sentences generated by G.

To support the formal definition of grammar we must elaborate the four component of the definition: terminal vocabulary, nonterminal vocabulary, production rule, and start symbol.

Terminal vocabulary, V_T, is the set of terminal elements with which the sentences of a language may be constructed. It is also called alphabet of the language. Elements of V_T will be denoted by lower case letters from the beginning of the Latin alphabet.

The nonterminal vocabulary, V_T, consists of elements which are only used in the derivation of a sentence; they never occur as such in the sentences of the language. Elements of V_N are indicated by upper case Latin letters and are called variables.

The production rules or productions of a grammar are ordered pairs of strings. They take the form $\alpha \rightarrow \beta$, where $\alpha \in V^+$ (V^+ is the set of all possible strings of vocabulary elements except the null-string, λ) and $\beta \in V^*$ (V* is the set of all possible strings of vocabulary elements). This means that string of elements α of positive length can be replaced by, or rewritten as, string of elements β, possibly λ. Such rules apply in any context, i.e. if α is part of a longer string $\gamma\alpha\delta$, then $\gamma\alpha\delta$ may be rewritten as $\gamma\beta\delta$ by the same rule. When a string is rewritten as another string by a single application of a production rule, we use the symbol $\Rightarrow$; thus $\gamma\alpha\delta \Rightarrow \gamma\beta\delta$. The latter string derives directly from the

former. If there are productions such that αl ⇒ α2, α2 ⇒ α3, … αn–l ⇒ αn, we may write α1 *⇒ αn, read "α1 derives αn".

The set of productions of a grammar is denoted by P; the set may also be described as a cartesian product. The set of all possible rules consists of all ordered pairs of strings which can be constructed in this manner; it may be denoted by $V^+ \times V^*$, the Cartesian product of V^+ and V*. The productions of a grammar are a subset of this product: some strings of V^+ may be replaced by some strings in V*. Thus $P \subset V^+ \times V^*$.

The start symbol of a grammar is denoted by *S* (originally for 'sentence'); it is a particular element of V_N.

3.3. Hierarchy of grammars

Chomsky (1959 a,b) devised a scheme for the classification of grammars which is now in general use. It is based on three increasingly restrictive conditions on the production rules.

First limiting condition: For every production α → β in P, |α| ≤ |β|. Thus the grammar contains no productions whose application would result in a decrease of string length.

Second limiting condition: For every production α → β in P, (1) α consists of only one variable, i.e. α ∈VN , and (2) β ≠ λ. The productions are of the form A → β, where β ∈V +.

Third limiting condition: For every production α → β in P, (1) α ∈VN , and (2) β has the form a or aB, where a ∈VT and β ∈VN . The rules are thus either of the form A → a or of the form A → aB.

With these limiting conditions, grammars may be classified in the following way.

Type-0 grammars are grammars which are not restricted by any of the limiting conditions. Their definition is simply that of 'grammar'; they are also called unrestricted rewriting systems. Productions are of the form α → β.

Type-1 grammars are grammars restricted by the first limiting condition. Productions have the form α → β, where |α| ≤ |β|. Type-1 grammars are also called context-

sensitive grammars (CSG) and it isa way to describe the syntax of natural language where it is indeed often the case that a word may or may not be appropriate in a certain place depending upon the context. They obviously constitute a subclass of type-0 grammars. In fact they are a strict subset of the set of type-0 grammars, for there are type-0 grammars which are not of type-1, namely, those grammars with at least one production where $|\alpha| > |\beta|$.

Type-2 grammars are grammars restricted by the second limiting condition. Productions have the form $A \rightarrow \beta$ where $\beta \neq \lambda$. Grammars of this type are called context-free grammars (CFG). The second condition implies the first: from $|\beta| \geq 1$ and $|A| = 1$ it follows that $|A| \leq |\beta|$. Context-free grammars are therefore context-sensitive, but the inverse is not true; the class of context-free grammars is a strict subset of the class of context sensitive grammars. Syntax of programming languages are presented in CFG.

Type-3 grammars are grammars restricted by the third limiting condition. Productions have the form $A \rightarrow a$ or $A \rightarrow aB$. These are regular grammars (in linguistic literature they are often called finite state grammars), In its turn the third limiting condition implies the second.

Therefore the class of regular grammars is a subclass of the class of context-free grammars; in fact it is a strict subset.

Language types may be defined according to the various classes of grammars. A type-3 grammar generates a regular language (or finite state language), a type-2 grammar generates a context-free language, a type-1 grammar generates a context-sensitive language, and a type-0 grammar generates a (recursively enumerable) language.

It does not follow, however, from the relations of inclusion which exist among the various types of grammars that corresponding languages are bound by the same relations of inclusion. We cannot exclude the possibility a priori that for every context-free grammar there might exist, an equivalent regular grammar. In that case all context-free languages might be generated by regular grammars, and consequently regular languages would not form a strict subset of context-free grammars. However in the following it will become apparent that the language types do show the same relations of strict inclusion as the grammar types: there are type-0 languages which are not context-sensitive, context-

sensitive languages which are not context-free, and context-free languages which are not regular. Figure 3.1., illustrates this hierarchical relation, called the Chomsky Hierarchy.

Every regular grammar is evidently context free. The statement that the grammars type 1 are at the same time type 0, is trivial, because all the grammars are type 0. The context free grammar, however, does not have to be necessarily type 1. Vanicek, et al. (2008)

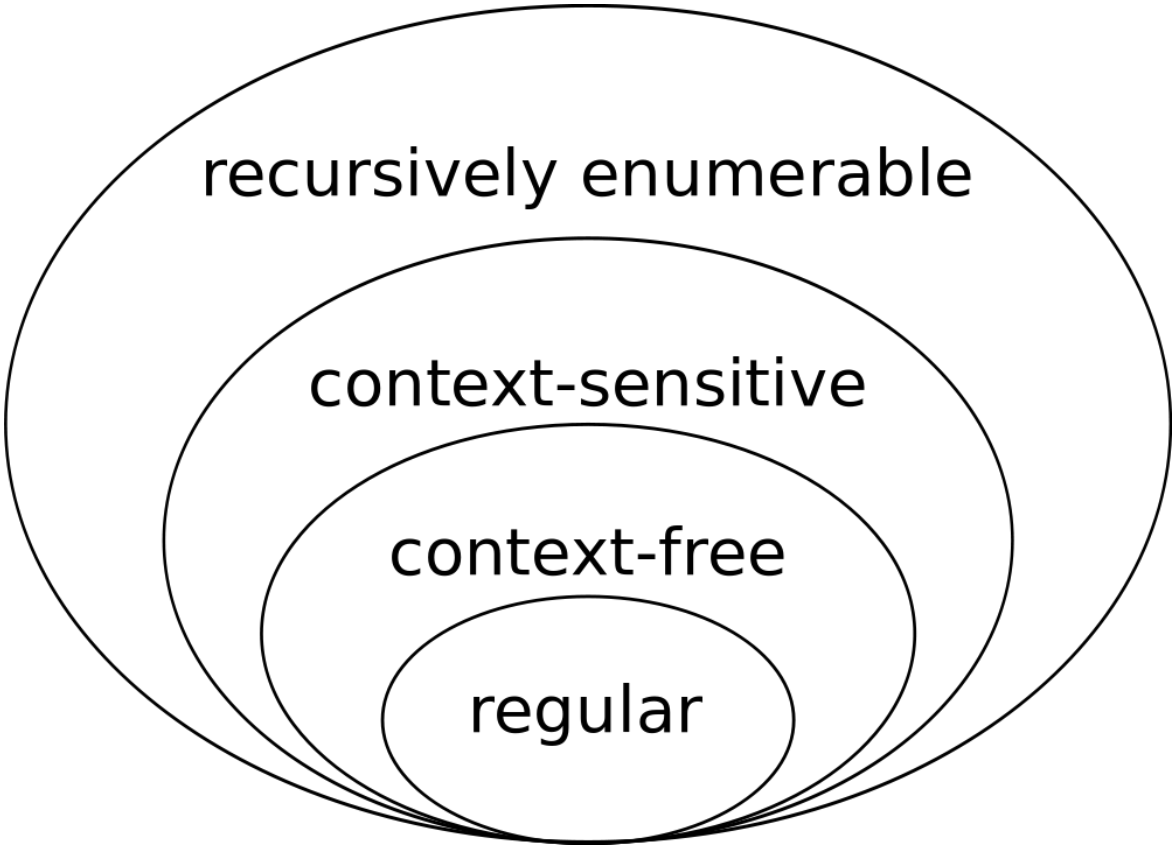


Figure 3.1. Chomsky Hierarchy of languages. Source: Chomsky, 1957

3.4.Programming languages

According to Watt (2004) programming languages are how people talk to computers. The computer would be just as happy speaking any language that was unambiguous. The reason we have high level languages is because people can't deal with machine language. The point of programming languages is to prevent our poor frail human brains from being overwhelmed by a mass of detail.

According to (Aho, et al 2007) programming languages are notations for describing computations to people and machines. And the world depends on programming languages, because all the software running on all the computers are written in some programming language.

3.5.Programming linguistics

Linguistics is the scientific study of human language including form, meaning and context. Likewise, for programming language there is a linguistics that studies its meaning, form and the languages meaning in a given context.

Watt(2004) defined programming linguistics as follows:

> We sometimes use the term programming linguistics to mean the study of programming languages. This is by analogy with the older discipline of linguistics,which is the study of natural languages. Both programming languages and natural languages have syntax (form) and semantics (meaning). However, we cannot take the analogy too far. Natural languages are far broader, more expressive, and subtler than programming languages. A natural language is just what a human population speaks and writes, so linguists are restricted to analyzing existing (and dead) natural languages. On the other hand, programming linguists can not only analyze existing programming languages; they can also design and specify new programming languages, and they can implement these languages on computers.(p. 3)

If we look at the history of programming languages, a lot of the best ones were languages designed for their own authors to use, and a lot of the worst ones were designed for other people to use.

When languages are designed for other people, it's always a specific group of other people: people not as smart as the language designer. So you get a language that talk down

to you. COBOL (COmmon Business Oriented Language) is the most extreme case, but a lot of languages are pervaded by this spirit (Watt, 2004).

It has nothing to do with how abstract the language is. C is pretty low-level, but it was designed for its authors to use, and that's why hackers like it.

Historical development of programming languages according to watt (2004):

> Today's programming languages are the product of developments that started in the 1950s. Numerous concepts have been invented, tested, and improved by being incorporated in successive programming languages. With very few exceptions, the design of each programming language has been strongly influenced by experience with earlier languages. The following brief historical survey summarizes the ancestry of the major programming languages and sketches the development of the concepts introduced in this book. It also reminds us that today's programming languages are not the end product of developments in programming language design; exciting new concepts, languages, and paradigms are still being developed, and the programming language scene ten years from now will probably be rather different from today's.
>
> Figure 3.2 summarizes the dates and ancestry of several important programming languages. This is not the place for a comprehensive survey, so only the major programming languages are mentioned.
>
> FORTRAN was the earliest major high-level language. It introduced symbolic expressions and arrays, and also procedures (''subroutines'') with parameters. In other respects FORTRAN (in its original form) was fairly low-level; for example, control flow was largely affected by

conditional and unconditional jumps. FORTRAN has developed a long way from its original design; the latest version was standardized as recently as 1997.

COBOL was another early major high-level language. Its most important contribution was the concept of data descriptions, a forerunner of today's data types. Like FORTRAN, COBOL's control flow was fairly low-level. Also like FORTRAN, COBOL has developed a long way from its original design, the latest version being standardized in 2002.

ALGOL60 was the first major programming language to be designed for communicating algorithms, not just for programming a computer. ALGOL60 introduced the concept of block structure, whereby variables and procedures could be declared wherever in the program they were needed. It was also the first major programming language to support recursive procedures. ALGOL60 influenced numerous successor languages so strongly that they are collectively called ALGOL-like languages.

FORTRAN and ALGOL60 were most useful for numerical computation and COBOL for commercial data processing. PL/I were an attempt to design a general-purpose programming language by merging features from all three. On top of these it introduced many new features, including low-level forms of exceptions and concurrency. The resulting language was huge, complex, incoherent, and difficult to implement. The PL/I experience showed that simply piling feature upon feature is a bad way to make a programming language more powerful and general-purpose.

A better way to gain expressive power is to choose an adequate set of concepts and allow them to be combined systematically. This was the design philosophy of ALGOL68. For instance, starting with concepts such as integers, arrays, and procedures, theALGOL68 programmer can declare an array of integers, an array of arrays, or an array of procedures; likewise, the programmer can define a procedure whose parameter or result is an integer, an array, or another procedure.

PASCAL, however, turned out to be the most popular of the ALGOL-like languages. It is simple, systematic, and efficiently implementable. PASCAL and ALGOL68 were among the first major programming languages with both a rich variety of control structures (conditional and iterative commands) and a rich variety of data types (such as arrays, records, and recursive types).

C was originally designed to be the system programming language of the UNIX operating system. The symbiotic relationship between C and UNIX has proved very good for both of them. C is suitable for writing both low-level code (such as the UNIX system kernel) and higher-level applications. However, its low-level features are easily misused, resulting in code that is unportable and unmaintainable.

PASCAL’s powerful successor, ADA, introduced packages and generic units –designed to aid the construction of large modular programs – as well as high-level forms of exceptions and concurrency. Like PL/I, ADA was intended by its designers to become the standard general-purpose programming language. Such a stated ambition is perhaps very rash, and ADA also attracted a lot of criticism.

(For example, Tony Hoare quipped that PASCAL, like ALGOL60 before it, was a marked advance on its successors!) The critics were wrong: ADA was very well designed, is particularly suitable for developing high-quality (reliable, robust, maintainable, efficient) software, and is the language of choice for mission-critical applications in fields such as aerospace.

We can discern certain trends in the history of programming languages. One has been a trend towards higher levels of abstraction. The mnemonics and symbolic labels of assembly languages are abstract away from operation codes and machine addresses. Variables and assignment abstract away from inspection and updating of storage locations. Data types abstract away from storage structures. Control structures abstract away from jumps. Procedures abstract away from subroutines.

Packages achieve encapsulation, and thus improve modularity. Generic units abstract procedures and packages away from the types of data on which they operate, and thus improve reusability.

Another trend has been a proliferation of paradigms. Nearly all the languages mentioned so far have supported imperative programming, which is characterized by the use of commands and procedures that update variables. PL/I and ADA support concurrent programming, characterized by the use of concurrent processes.

However, other paradigms have also become popular and important.

Object-oriented programming is based on classes of objects. An object has variable components and is

equipped with certain operations. Only these operations can access the object's variable components. A class is a family of objects with similar variable components and operations. Classes turn out to be convenient reusable program units, and all the major object-oriented languages are equipped with rich class libraries.

The concepts of object and class had their origins in SIMULA, yet another ALGOL-like language. SMALLTALK was the earliest pure object-oriented language, in which entire programs are constructed from classes.

C++ was designed by adding object-oriented concepts to C. C++ brought together the C and object-oriented programming communities, and thus became very popular. Nevertheless, its design is clumsy; it inherited all C's shortcomings, and it added some more of its own.

JAVA was designed by drastically simplifying C++, removing nearly all its shortcomings. Although primarily a simple object-oriented language, JAVA can also be used for distributed and concurrent programming. JAVA is well suited for writing applets (small portable application programs embedded in Web pages), as a consequence of a highly portable implementation (the Java Virtual Machine) that has been incorporated into all the major Web browsers. Thus JAVA has enjoyed a symbiotic relationship with the Web, and both have experienced enormous growth in popularity. C# is very similar to JAVA, apart from some relatively minor design improvements, but its more efficient implementation makes it more suitable for ordinary application programming.

Functional programming is based on functions over types such as lists and trees. The ancestral functional language was LISP, which demonstrated at a remarkably early date that significant programs can be written without resorting to variables and assignment.

ML and HASKELL are modern functional languages. They treat functions as ordinary values, which can be passed as parameters and returned as results from other functions. Moreover, they incorporate advanced type systems, allowing us to write polymorphic functions (functions that operate on data of a variety of types).

ML (like LISP) is an impure functional language, since it does support variables and assignment. HASKELL is a pure functional language.

As noted in Section 1.1.1, mathematical notation in its full generality is not implementable. Nevertheless, many programming language designers have sought to exploit subsets of mathematical notation in programming languages.

Logic programming is based on a subset of predicate logic. Logic programs infer relationships between values, as opposed to computing output values from input values. PROLOG was the ancestral logic language, and is still the most popular.

In its pure logical form, however, PROLOG is rather weak and inefficient, so it has been extended with extra-logical features to make it more usable as a programming language.

Programming languages are intended for writing application programs and systems programs. However, there are other niches in the ecology of computing.

An operating system such as UNIX provides a language in which a user or system administrator can issue commands from the keyboard, or store a command script that will later be called whenever required. An office system (such as a word processor or spreadsheet system) might enable the user to store a script (''macro'') embodying a common sequence of commands, typically written in VISUAL BASIC.

The Internet has created a variety of new niches for scripting. For example, the results of a database query might be converted to a dynamic Web page by a script, typically written in PERL. All these applications are examples of scripting. Scripts (''programs'' written in scripting languages) typically are short and high-level, are developed very quickly, and are used to glue together subsystems written in other languages. So scripting languages, while having much in common with imperative programming languages, have different design constraints. The most modern and best-designed of these scripting languages is PYTHON. (P.6-10)

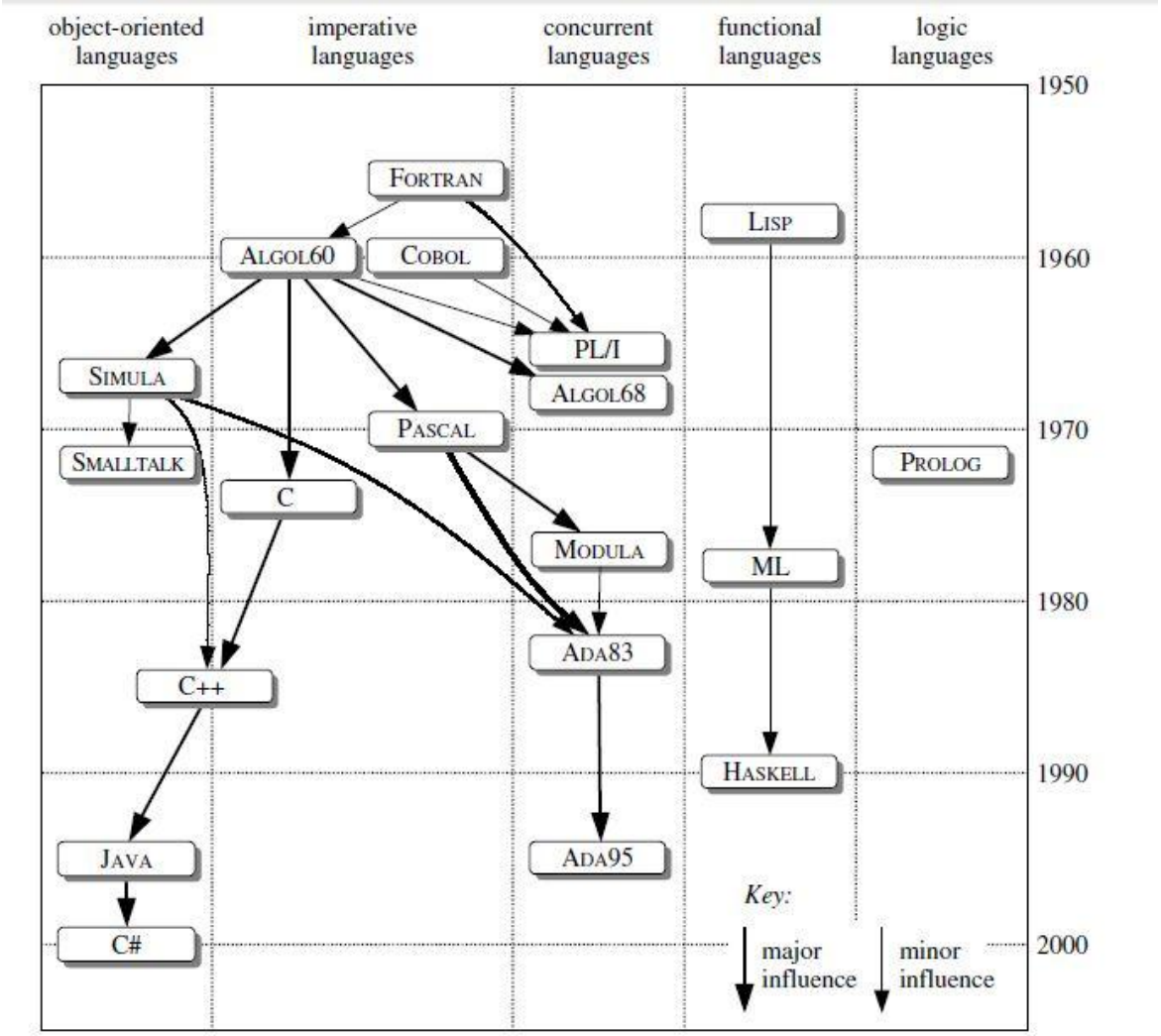


Figure 3.2 Dates and ancestry of major programming languages. Source: Watt (2004)

3.6.Language translator and processor

In order to solve a problem using computer; we must write instruction in programming language, translate in to a form understandable by the computer and execute instruction. To support this process of problem solving; we will use tools like language processor and translator.

Watt (2004) defined language processer as "Any system for processing programs – executing programs, or preparing them for execution – is called a language processor. Language processors include compilers, interpreters, and auxiliary tools like source-code editors and debuggers."

Now a day there are plenty of language processors which range from notepad to IDE (Integrated Development Environment). Some of these have advanced features to aid the programmer, like .Net's intellisense, error report of turbo C++. Availability of such tools had made programming easier and comfortable, even for beginners.

Before a program can be run, it first must be translated into a form in which it can be executed by a computer. The software systems that do this translation are called language translators. Based on the way of the translation we have two kinds of translator; Compiler and Interpreter.

Compiler, based on Aho, et al. (2007), is a program that can read a program in one language- the source language - and translate it into an equivalent program in another language - the target language. An important role of the compiler is to report any errors in the source program that it detects during the translation process. If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs. Compilation is presented in figure 3.3

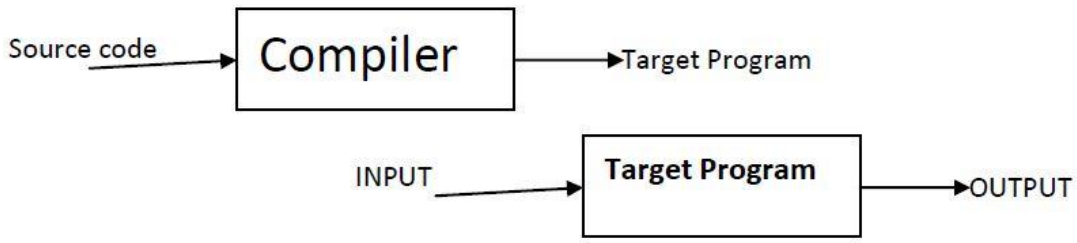


Figure 3.3 The compilation process. Source: own

Interpreter, according to Aho, et al. (2007), is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

Figure 3.4 the process of interpretation. Source: own

The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs. An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

Compiler is a software system which has five or more phases. Those phases are grouped as front end and back end, also called analysis and synthesis respectively. The analysis part is responsible for braking down the source code in to a serious of lexeme (a sequence of characters from the input that match a pattern) and produce tokens (symbolic names for the entities that make up the text of the program) representation of the language. It is also responsible to provide sophisticated error message related to syntax violation and semantic interpretation, which will be used by the programmer for debugging. The synthesis part is responsible for generating the target program and make use of the symbol table and the intermediate representation generated by the previous part.

Aho, et al. (1986), defined the function of these parts as follows:

> The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it

in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.
The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table.

The analysis part consists of lexical analyzer, syntax analyzer, semantic analyzer and intermediate code generator; the synthesis part is composed of machine-independent intermediate representation, code generator, target-machine code and machine-dependent code optimizer. A typical decomposition of compiler into phases is shown in Fig. 3.5

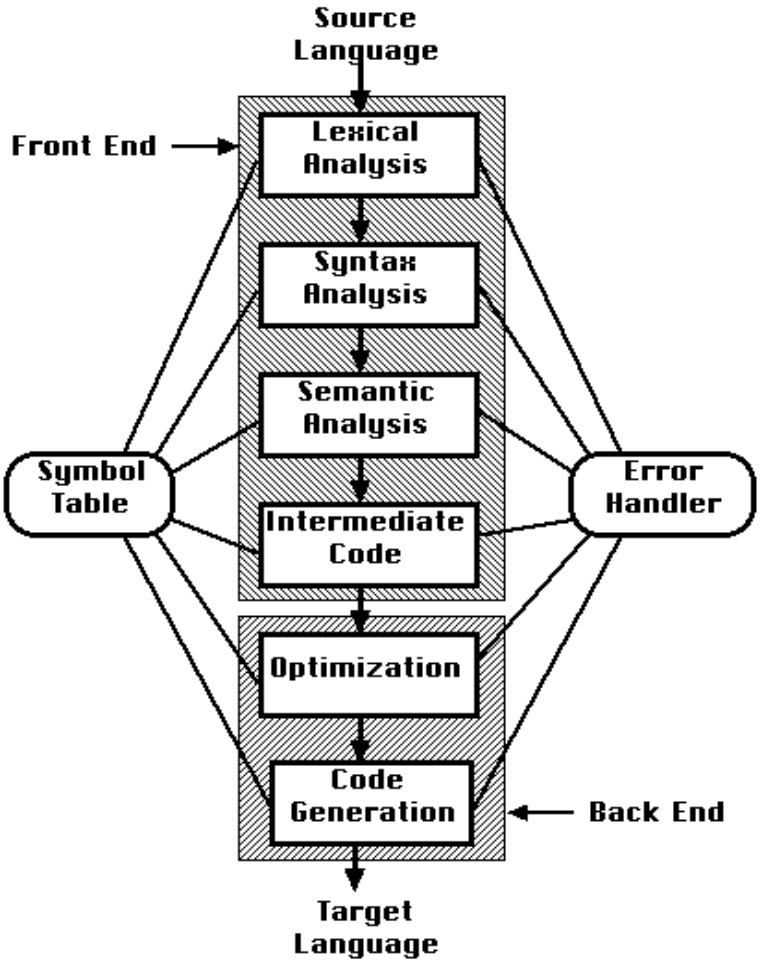


Figure 3.5 phases of a compiler. Source: Aho, et al. 2007

The first phase of a compiler is called *lexical analysis* or *scanning.* The lexical analyzer reads the stream of characters making up the source program and groups the

characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form (token-name, attribute-value) that it passes on to the subsequent phase, syntax analysis. In the token, the first component token-name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token. Information from the symbol-table entry 'is needed for semantic analysis and code generation.

The second phase of the compiler is syntax analysis or parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.

The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we

can think of as a program for an abstract machine. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.
The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.
The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the judicious assignment of registers to hold variables. Aho, et al. (2007)

3.7.Characters and Character sets

Characters (The Linux Information Project, 2007) are the basic symbols that are used to write or print a language. For example, the characters used by the English language consist of the letters of the alphabet, numerals, punctuation marks and a variety of symbols (e.g., the ampersand, the dollar sign and the arithmetic symbols).

According to Linux information project (The Linux Information Project, 2007) characters are fundamental to computer systems. They are used for input (e.g., through the keyboard or through optical scanning) and output (e.g., on the screen or on printed pages), writing programs in programming languages, as the basis of some operating systems (such as Linux) which are largely collections of plain text(i.e., human-readable character) files and for the storage and transmission of non-character data (e.g., the transmission of images by e-mail using base64).

A general definition of character set is given by Linux information project (The Linux Information Project, 2007), as a collection of characters that is used to write a particular language. Most languages have a single character set, and similar character sets are often used by a number of languages (e.g., variants of the Roman alphabet are used to write English, Spanish, Finnish, Dutch, etc.)

Webopedia, www.webopedia.com, defined character set as a defined list of characters recognized by the computer hardware and software. Each character is represented by a number. There are different character sets: ASCII (American standard code for information interchange), uses the numbers 0 through 127 to represent all English characters as well as special control characters; ISO 8859-1(Latin-1), are similar to ASCII but they contain additional characters for European languages; Unicode, which contains all of the characters commonly used in information processing.

ASCII functions as a common denominator between computers that otherwise have nothing in common. It works by assigning standard numeric values to letters, numbers, punctuation marks and other characters such as control codes. Bob Bemer, who was instrumental in ASCII's development, said "we had over 60 different ways to represent characters in computers. It was a real Tower of Babel". ASCII is contained within 2^7, or 128 characters. There's room in ASCII for upper and lowercase English, American English punctuation, digits and a few control characters. Although very primitive, it's important to note ASCII is the one common denominator contained in all the other common character sets - so the only means of interchanging data across all major languages (without risk of character mapping loss) is to use ASCII (or have all sides understand Unicode).(ASCII).

ISO 8859-1 encodes what it refers to as "Latin alphabet no. 1," consisting of 191 characters from the Latin script. This character-encoding scheme is used throughout The Americas, Western Europe, Oceania, and much of Africa. It is also commonly used in most standard Romanization of East-Asian languages. Each character is encoded as a single eight-bit code value. These code values can be used in almost any data interchange system to communicate in the following European languages (with a few exceptions due to missing characters)(ISO/IEC 8859-1:1998.).

Unicode is a 16-bit character set which contains all of the characters commonly used in information processing. Approximately 1/3 of over 1 million possible code points are still unassigned, to allow room for adding additional characters in the future. It is not a technology in itself. Sometimes people misunderstand Unicode and expect it to 'solve' international engineering, which it doesn't. It is an agreed upon way to store characters, a standard supported by members of the Unicode Consortium.

The fundamental idea behind Unicode is to be language-independent, which helps conserve space in the character map - no single character is assumed to identify a language in itself. Just like a character "a" can be a French, German or English "a" even if they have different meanings, a particular Han ideograph might map to a character used in Chinese, Japanese and Korean. Sometimes native speakers of these languages misunderstand Unicode as not "looking" correct in Japanese for example, but that's intentional - appearance should reside in the font as an artistic issue, not the code point as an engineering issue. Although it's technically possible to ship one font which covers all Unicode characters, it would have very limited commercial use, since end-users in Asia will expect fonts dedicated and designed to look correct in their language.

This language-independence also means Unicode does not imply any sort order. The older 8-bit and DBCS character sets usually contain a sort order, but this means they had to create a new character set to change the sort order, which makes a mess out of data

> interchange between languages. Instead, Unicode expects the host operating system to handle sorting, as the Win32 NLS APIs do.
> (http://www.microsoft.com/typography/unicode/cs.htm)

Computer programs are still written using part of 128 characters, the ASCII characters, or 256 characters, the ISO-latin-1 characters. But in a few years, using all the Unicode characters in programs may be standard, which consists of 109384 encoded characters in Unicode 6.0 and 1114112 total codes which include 1111998 characters in 17 planes, 2048 surrogates and 66 non characters (The Unicode Consortium, 2012). Mathematics is a language that uses many more tokens than most of the current programming languages, both dedicated symbols and letters from many alphabets. It can describe concepts extremely tersely, since the greater the range of characters a language has, the terser it can be written.

Committed programmers are continually looking for ways to make programs terser, yet still readable. They choose languages and tools that enable such terseness, so programming languages evolved, into the 2GL (second generation language: assembly language), the 3GL, and the visual 4GL. But 4GL's were limited in their scalability and readability. It's easier to write a program in a 4GL than a 3GL, but more difficult to read and debug it. So some used IDE's, supplementing 3GL's with visual aids. Others looked for a more productive language, so terser languages, such as Perl, Python, and Ruby, became popular. Regular expressions are a successful attempt at tersity, now used by many languages, but many consider them unreadable. The K programming language, used by financial businesses, could be the tersest language ever invented. It only uses ASCII symbols, but overloads them profusely. However, the price is the inability to give different precedence to the operators, so everything unbracketed is evaluated from the right. The terseness of present-day programming languages is derived from maximizing the use of grammar, the different ways characters can be combined.

Operator overloading in C++ was a similar attempt at tersity. Programmers could define meanings for some combinations of the 35 ASCII symbols. Although programs became terser, they were more difficult to understand because of the unpredictable meaning of these symbols in different code contexts, and they were eventually dropped in

Java. The problem wasn't with operator overloading itself, but with the uncontrolled association of meanings with each operator. Eventually certain meanings would have become generally accepted, the others falling into disuse, but this would have taken many years, with too many incompatible uses produced in the meantime. If there was such a problem with a few dozen operators, what hope would there be for the hundreds of unused Unicode symbols? If programmers were allowed to overload them with any meaning, the increase in program tersity would be at the cost of readability.

Perhaps adding the many Unicode symbols to programming languages would enable terser programs to be written. Although some Unicode symbols will have an obvious meaning, such as some mathematical symbols, most would have no meaning that could be transferred easily to the programming context. To retain readability of programs in a terse language, the meanings of the Unicode symbols would have to be carefully controlled by the custodians of that language. They would activate new Unicode symbols at a gradual pace only, with control of their meanings, after carefully considering existing use of the symbols.

Programming languages do, however, already allow Unicode characters in some parts of their programs. The contents of strings and comments can use any Unicode character. User-defined names can use all the alphabetic letters and symbols in utf-8(Unicode transformation format) character set, and because there is already agreed meanings for combinations of these, derived from their respective natural languages, we can increase tersity while keeping readability. But the core of the language, the grammar keywords and symbols, and names in supplied libraries, still only use ASCII characters. Perhaps some programmers use non-Latin characters wherever they can in their programs.

Programmers from cultures not using the Latin alphabet won't be motivated to use their own alphabets in user-defined names when they don't with pre-supplied names, such as keywords or standard libraries. Often, most of the names in a program are from libraries standard to the language. To trigger the widespread use of Unicode characters from non-ASCII alphabets in programs, the pre-supplied names must also be in those alphabets. And this could easily be done. The grammar of a language and its vocabulary are two different concepts. A programming language grammar could conceivably have many vocabulary options. Almost all programming languages only have English. Other vocabularies could

be based on other natural languages. A Spanish vocabulary plugged into a certain programming language would have Spanish names for the keywords, modules, classes, methods, variables, etc.

Computer software nowadays is internationalized, web pages are, and most programming languages enable internationalized software. But the languages themselves and their libraries are not internationalized. An internationalized programming language would enable a fully functional program to be written totally in a natural language of one's choice. Not only could all user-defined names be specified in any alphabet, but also all keywords and names in the standard libraries would be available in many natural languages.

Amharic is a language spoken in Ethiopia and Eritrea. It is the second most spoken Semitic language next to Arabic. It uses Unicode characters in range of 1200-137F as main set of characters and extended ranges are available time to time to support the symbols and dialects in the language. The author a native speaker of the language will design a grammar for a Pascal like language in Amharic. The main purpose of the author is to design a programming language suitable to teach programming for Amharic speaking student and to implement it in beta version. So at the end of the study there will be language grammar in BNF to describe the vocabulary and syntax of the intended language, a compiler front end which could translate the given language program in to three address code, sample test program and sample output in three address code.

<letter> = 'ሀ' | 'ሁ' | 'ሂ' | 'ሃ' | 'ሄ' | 'ህ' | 'ሆ' | 'ለ' | 'ሉ' | 'ሊ' | 'ላ' | 'ሌ' | 'ል' | 'ሎ' | 'ሐ' | 'ሑ' | 'ሒ' | 'ሓ' | 'ሔ' | 'ሕ' | 'ሖ' | 'መ' | 'ሙ' | 'ሚ' | 'ማ' | 'ሜ' | 'ም' | 'ሞ' | 'ሠ' | 'ሡ' | 'ሢ' | 'ሣ' | 'ሤ' | 'ሥ' | 'ሦ' | 'ረ' | 'ሩ' | 'ሪ' | 'ራ' | 'ሬ' | 'ር' | 'ሮ' | 'ሰ' | 'ሱ' | 'ሲ' | 'ሳ' | 'ሴ' | 'ስ' | 'ሶ' | 'ሸ' | 'ሹ' | 'ሺ' | 'ሻ' | 'ሼ' | 'ሽ' | 'ሾ' | 'ቀ' | 'ቁ' | 'ቂ' | 'ቃ' | 'ቄ' | 'ቅ' | 'ቆ' | 'በ' | 'ቡ' | 'ቢ' | 'ባ' | 'ቤ' | 'ብ' | 'ቦ' | 'ተ' | 'ቱ' | 'ቲ' | 'ታ' | 'ቴ' | 'ት' | 'ቶ' | 'ቸ' | 'ቹ' | 'ቺ' | 'ቻ' | 'ቼ' | 'ች' | 'ኾ' | 'ኀ' | 'ኁ' | 'ኂ' | 'ኃ' | 'ኄ' | 'ኅ' | 'ኆ' | 'ኘ' | 'ኙ' | 'ኚ' | 'ኛ' | 'ኜ' | 'ኝ' | 'ኞ' | 'አ' | 'ኡ' | 'ኢ' | 'ኣ' | 'ኤ' | 'እ' | 'ኦ' | 'ከ' | 'ኩ' | 'ኪ' | 'ካ' | 'ኬ' | 'ክ' | 'ኮ' | 'ኸ' | 'ኹ' | 'ኺ' | 'ኻ' | 'ኼ' | 'ኽ' | 'ኾ' | 'ወ' | 'ዉ' | 'ዊ' | 'ዋ' | 'ዌ' | 'ው' | 'ዎ' | 'ዐ' | 'ዑ' | 'ዒ' | 'ዓ' | 'ዔ' | 'ዕ' | 'ዖ' | 'ዘ' | 'ዙ' | 'ዚ' | 'ዛ' | 'ዜ' | 'ዝ' | 'ዞ' | 'ዟ' | 'ዠ' | 'ዡ' | 'ዢ' | 'ዣ' | 'ዤ' | 'ዥ' | 'ዦ' | 'ዧ' | 'የ' | 'ዩ' | 'ዪ' | 'ይ' | 'ዬ' | 'ዮ' | 'ደ' | 'ዱ' | 'ዲ' | 'ዳ' | 'ዴ' | 'ድ' | 'ዶ' | 'ጀ' | 'ጁ' | 'ጂ' | 'ጃ' | 'ጄ' | 'ጅ' | 'ጆ' | 'ገ' | 'ጉ' | 'ጊ' | 'ጋ' | 'ጌ' | 'ግ' | 'ጎ' | 'ጠ' | 'ጡ' | 'ጢ' | 'ጣ' | 'ጤ' | 'ጥ' | 'ጦ' | 'ጨ' | 'ጩ' | 'ጪ' | 'ጫ' | 'ጬ' | 'ጭ' | 'ጮ' | 'ጰ' | 'ጱ' | 'ጲ' | 'ጳ' | 'ጴ' | 'ጵ' | 'ጶ' | 'ጷ' | 'ጸ' | 'ጹ' | 'ጺ' | 'ጻ' | 'ጼ' | 'ጽ' | 'ጾ' | 'ፀ' | 'ፁ' | 'ፂ' | 'ፃ' | 'ፄ' | 'ፅ' | 'ፆ' | 'ፈ' | 'ፉ' | 'ፊ' | 'ፋ' | 'ፌ' | 'ፍ' | 'ፎ' | 'ፐ' | 'ፑ' | 'ፒ' | 'ፓ' | 'ፔ' | 'ፕ' | 'ፖ' | 'ቐ' | 'ቒ' | 'ቓ' | 'ቔ' | 'ቕ' | 'ቖ' | 'ቘ' | 'ቚ' | 'ቛ' | 'ቜ' | 'ቝ'

Figure 3.6: alphabets in Amharic language. Source: own

3.8.Open Source versus Closed Source

As discussed in (The Linux Information Project, 2007)

One of the most intensely debated topics in the computer field continues to be the relative merits of open source and closed source software. The former is software for which the source code is freely available (i.e., at no cost and easily accessible) for anyone to use for any purpose, including studying, modifying, extending, giving away or even selling. Although there are some philosophical differences behind the movements, in most cases open source software is basically the same as free software.

The latter provide their source code either with the compiled software or by allowing it to be downloaded from the Internet. The GNU General Public License (GPL), the most widely used free software license, actually makes it a legal requirement that the source code for all software released under it be made freely available to all users.

Closed source software is software for which the source code is kept secret. Most proprietary (i.e., commercial) software is closed source. There are several reasons that developers of proprietary software take great pains to keep their source code secret, including the concerns that:

1. Other developers might copy some of their code and use it in other programs.
2. Hackers will find vulnerabilities in the code that will enable them to to develop viruses, spyware or other malware (i.e., malicious software) for it.
3. Public disclosure of the source code could expose

its developers to charges that some of the code was plagiarized from other programs.

4. Customers will try to modify the source code, resulting in new problems that could be difficult for either the customers or developer to correct.

5. The source code could be used as evidence in legal proceedings, particularly those related to whether the developer has been complying with certain legislation or court decisions.

6. The source code could contain unfavorable comments inserted by programmers about their employer, customers or competitors. Comments are words, phrases, sentences or paragraphs that are interspersed in source code but which do not affect the operation of the code. Their official purpose is to document the code and explain it to other programmers (who may have to repair or revise the code at some future date), although they are frequently used for other purposes as well. It can be difficult to find and remove potentially offensive comments because of the great length of the code for large programs and the subtlety with which some comments are written.

There are also reasons for not keeping source code secret and for allowing, or even encouraging, others to view and study it. Advocates of open source point out that this approach makes it possible for a much larger and more diverse set of qualified people to examine the source code, thus resulting in the discovery of more bugs and providing more and better suggestions for improvements and extensions.

That this approach has been extremely successful is evidenced by the rapid improvements in performance of numerous open source programs, many of which are equal to or superior to their closed source counterparts. One of the most outstanding examples is the Apache web server, which is currently, hosts more than 70 percent of all web sites on the Internet.

Undoubtedly the most famous example is Linux, the use of which is continuing to grow rapidly for a wide range of applications, including supercomputers, enterprise computing systems, personal computers and embedded systems. Another example is the GNU Compiler Collection (GCC), which contains very highly rated and widely used compilers for C, C++, FORTRAN, Java and other programming languages. In fact, the concept of open source is so appealing that there are currently thousands of open source projects in various stages of development.

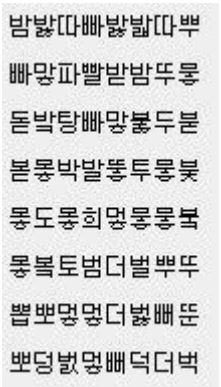
3.9.Sample Programming that uses Unicode

Even though there are many programming language now a day, most of them share that they are based on English. And with the available tools and computing power developing language become much easier than the past when it comes to English based languages.

Non-English-based programming languages are computer programming languages that, unlike known programming languages, do not use keywords taken from, or inspired by, the English vocabulary.

Algol 68 is a powerful, high-level, general-purpose programming language ideally suited to modern operating systems. It was the first to publish the standard in many languages, and the standard allowed the internationalization of the language itself. On December 20, 1968, the "Final Report" (MR 101) was adopted by the Working Group 2.1, then subsequently approved by the General Assembly of UNESCO's IFIP for publication. Translations of the standard were made for Russian, German, French, Bulgarian, and then later Japanese. The standard was made available in Braille. It went on to become the GOST/ГОСТ-27974-88 standard in the Soviet Union[2]. In English, its *reverent* case statement reads **case** ~ **in** ~ **out** ~ **esac**. In Cyrillic, this reads **выб** ~ **в** ~ **либо** ~ **быв** .

Aheui[3](아희) is an esoteric programming language, a computer programming language designed to experiment with weird ideas, to be hard to program in, or as a joke, rather than for practical use, first ever to be designed for the Hangul/Hangeul which is the Korean alphabet. It is functionally a family of INTERCAL, Brainfuck and Befunge but many of the language's concept is derived from Befunge[4], except the fact that it has no instruction for self-modifying, and that it has 26 stacks and one queue. The code of Aheui is written in UTF-8 encoding. Only Hangul syllables (from AC00 to D7A3) are recognized as a command; others are ignored.



Figuer 3.7: Sample Hello World program written in Aheui

[2] "GOST 27974-88 Programming language ALGOL 68 - Язык программирования АЛГОЛ 68" (in Russian) (PDF).
[3] http://puzzlet.springnote.com/pages/219154.xhtml
[4] http://esolangs.org/wiki/Befunge

Jeem (ج)– Arabic programming language, based on C and Pascal with simple graphics implementation. This language is developed with the aim of making programming easy for Arab students, and uses the utf-8 code range from 0600 to 06FF. Since this language is written for Arabic user the documentation is also in Arabic so it is hard to learn for non-Arabic speaker.

```
)*
----------------------------------------------------------------------------------------------
برنامج بلغة ج لحل معادلة بمجهول واحد من الدرجة الثانية في مجال الأعداد الحقيقية
جميع الحقوق محفوظة للمؤلف: د. محمد عمار السلكة ، 1420 هـ - 2000 م
----------------------------------------------------------------------------------------------
*(
!! الشكل العام للمعادلة آ س^2 + ب س + ج =0
المتحول آ، ب، ج : حقيقي
المتحول م، س1، س2 : حقيقي
أكرر طالما صواب
    }
        أكتب "أدخل آ ب ج (أدخل 0 قيمة لـ آ كي تنهي البرنامج) : "
        أقرأ آ ، ب ، ج
        إذا آ = 0 أنتهي
        أكتب " المعادلة: " ، آ ، " س^2 "
        إذا ب > 0 أكتب " + "
        أكتب ب ، " س "
        إذا ج > 0 أكتب " + "
        أكتب ج ، " = 0" ، سطر
        أجعل م = ب^2 - 4× آ × ج
        أكتب " المميز = "، م ، سطر
        إذا (م<0)
            أكتب "المعادلة مستحيلةالحل !"، سطر
        وإلا
            }
            أجعل س1 = (-ب + جذر(م)) ÷ (2×آ)
            أجعل س2 = (-ب - جذر(م)) ÷ (2×آ)
            أكتب " حل المعادلة: "،سطر
            أكتب "س1 = "، س1 ، سطر
            أكتب "س2 = " ، س2 ، سطر
                {
                    {
```

Figure 3.8: A program to solve a quadratic equation in Jeem

AMMORIA(in Arabic) is an object oriented programming language uses Arabic words instead of English words, to make learning programming for Arab children easy and fast, it's planned to support Urdo and Farsi too, AMMORIA has its won IDE and Visual tools.

Chinese Programming language, are languages which use Chinese characters from Unicode character set. And consists Chinese BASIC, Easy Programming Language, ChinesePython, Mama: is an educational programming language and was designed to help young students develop 3D animations and games, and RoboMind: is another educational programming language available in many non-English languages, including Chinese and it introduces computer science and robotics.

Now a day there are tones of programming languages in non-English based language even if, it is not comparable the number of programming language available in English. Most of the language has been developed, in USA, UK, Canada and Australia. And English is their native language or second language, so English become dominant in computing area. This has difficulty for those countries which use English by no means.

Ethiopia is a history rich country with its own script and calendar. Since it has more than eighty ethnic groups, there are languages which are Latin based and Geez based (A language dominant in Orthodox Church). And It uses Amharic, which is geez based, as its official language, while regions can implement their own official language, but academics is unified in English as primary teaching language. Even if Amharic is spoken in almost all the nation of Ethiopia and part of Eretria there is no or very little effort to make this language a computing language. As a matter of fact students have to learn two things at the same time to learn programming, English and Programming language.

Geez#, a fidel based C#, which it can be considered as the localized version of C#. It is based upon the Amharic alphabet and uses Semitic-based “Geez” characters native to Ethiopian languages. It is a great breakthrough for fidel to be used in computing but Geez# is mainly geez; a predominant language used in Ethiopian Orthodox Church and ancestor of Amharic. The problem of geez as a language for programming is that it is very difficult than English for most youngster; because the Church is not dictating the direction of education after the downfall of the Imperial regime and geez is not thought by force.

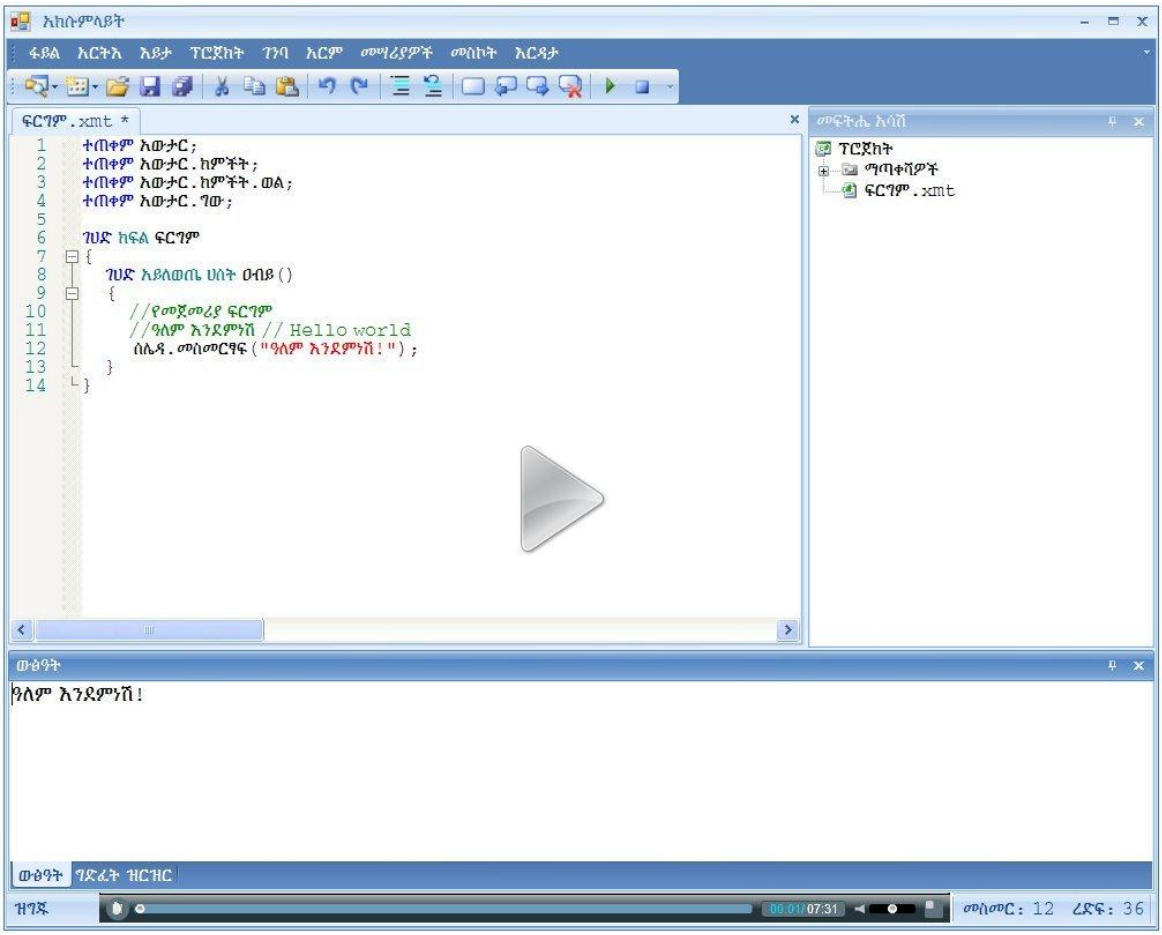


Figure 3.9 a hello world program in Geez# with AxumLight IDE. Source: Meliyu

This thesis will solve the problem of Geez# by implementing open source Jempas, a Pascal like language, totally based on Amharic. It will also provide the grammar for the language, which could enable teaching formal language theory, compiler construction and programming language much easier than now for Amharic speaking students.

4. Practical Part

4.1.BNF grammar for the language

The first step in developing a programming language is to design the grammar. The grammar is the definition or specification of the language and any grammar in BNF consists of start symbol. In my grammar the start symbol is *program* and in the grammar there are terminal and non-terminal symbols. Terminal of the grammars are composed of alphabet of the language and they make up the keywords in the language developed. Part of the grammar is enlisted here and explained, full of the grammar can be found in appendix.

The rule *<program> = 'ፕሮግራም' 'ዋና' <NAME> '(' ')' '{' [<var_part>] [<fun_part>] [<stat_part>] '}'* is the first rule, and consists of *<program>,* the start symbol*; 'ፕሮግራም' and 'ዋና'*, keywords, ; *<NAME>, [<var_part>], [<fun_part>]and [<stat_part>]*, non-terminals; and *'(', ')', '{' and '}'*, terminals. Angle brackets < > is used to represent non-terminals, square brackets [ ] is used to represent optional component of the grammar and single quotation is used to represent terminals in the grammar. Based on this rule a program is syntactically correct if it has the keyword *ፕሮግራም* followed by keyword *ዋና* and then the name of the program, opening and closing bracket, opening curl bracket, optional parts for variable declaration, function part and statement part, and terminated by closing curl bracket.

The rule *<var_part> = ['ማስቀመጫ' (<var_decl>)+]** is the next rule I want to explain due to some additional grammar symbols in it. The symbol + in grammar means that at least one occurrence of the symbol must occur. And the other symbol in this rule is the asterisk or *, which is to mean zero or more occurrence. So this rule explains that variable part could be empty or can be repeated where as if a variable part exist there should be at least one variable declaration.

The rule *<type> = 'ለቁጥር' | 'ለሀረግ'| 'ለእውነትህሰት'| 'ባዶ'* is the last rule I will be covering in my explanation. Here we have met a new symbol, |, it is to mean selection. One of the listed options could be used as a substitute for the non-terminal symbol *<type>*. And from the rule above one can learn there are four data types in my program.

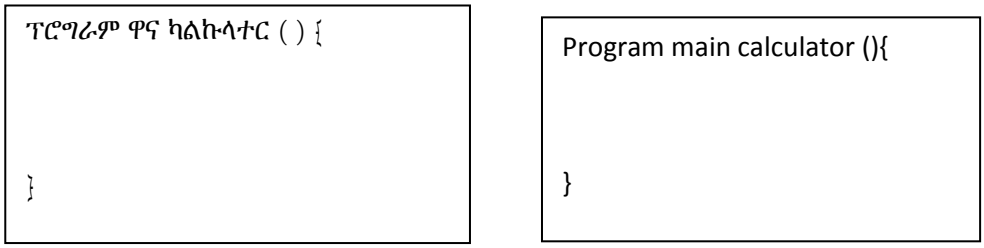


Figure 4.1 The simplest possible program in Jempas. Source: own

4.2.Source code

Source code is a text listing of commands to be compiled or assembled into an executable computer program. Source code (also referred to as source or code) is the human readable version of software as it is originally written (i.e., typed into a computer) by a human in plain text (i.e., human readable alphanumeric characters). It contains variable declarations, instructions, functions, loops, and other statements that tell the program how to function. And source code in Jempas should give a clue even for non-programmer Amharic language speakers. Jempas is writen in c on Linux and well commented to help any programmer understand, add future(s) or modify it, to be open source and will be available in GNU general public licenses.

The term software refers to all operating systems, application programs and data that are used by products containing microprocessors (also called processors or central processing units). Such products include not only personal computers but also a vast array of other products, such as aircraft electronic systems, railway signaling systems, industrial robots, electronic medical equipment, space vehicle guidance systems, electronic cameras and even simple electronic toys.

Source code can be written in any of the hundreds of programming languages that have been developed. Some of the most popular of these are C, C++, Cobol, Fortran, Java, Perl, PHP, Python and Tcl/Tk.

There are many programs that can be used for writing source code in the desired programming language, ranging from simple, general purpose text editors (such as vi or gedit on Linux or Notepad on Microsoft Windows) to integrated development environments (such as Visual C++ on Microsoft Windows or the cross-platform Eclipse

Platform for constructing and running integrated software-development tools). After writing, the source code is saved in a single file or, more commonly, in multiple files, with the number of files depending on such factors as the programming language and the size of the project. For Jempas It is advisable to use Unicode or utf aware text editor to write the source code in UNIX/Linux like environment. Therefore, Kate, Gedit and Vi are recommended.

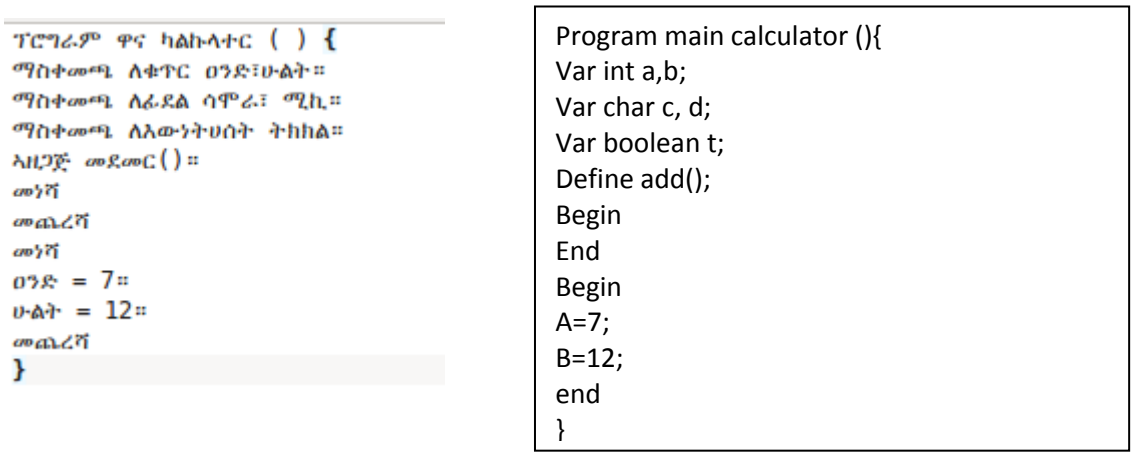


Figure 4.2 Structure of Jempas program (Left) and its literal translation(Right)

4.3.Lexical Analysis

Lexical analyzer or scanner is part of a compiler which scans characters in the source code and identifies them in to tokens. It removes whitespaces separating lexemes in the source program. The source code is written in Amharic in any utf supported text editor. To read the source file properly we have to handle wide character. Wide character is defined in c header file wchar.h.

To provide main function of a lexer, Jempas's scanner defined three public functions, get_prog(), get_token(), and get_lexem(). It reads the content in the input file in to buffer using get_prog(), and group meaning full character sequence as a lexeme using get_lexem() and it returns token for each lexeme in the source program. Those functions declared in lexer.h file externally to make them accessible from any file which has included lexer.h.

The setlocale() function have been used throughout the project to assist on the output statement of a wide character format Amharic text. It is defined under the header file locale.h. setlocale sets the program's current local for the category specified in parameter of setlocal(category, locale). To make a program portable for all locales one can call setlocale(LC_ALL, ""); that is what this project also preferred to use.

The lexer, to achieve its goal of reading a sequence of character, grouping them in to a meaning full lexeme and returning a token; it makes use of keytoktab.c file. This file consists of the definition of the lex2tok (), key2tok () and tok2lex () returning token, token and lexeme. It is in keytoktab.c the value of all token are defined as a pair of lexeme and token. For those lexemes which are more than single character long and are not in ASCII range a constant integer value greater than 256 is assigned in keytoktab.h with the declaration of lex2tok(), key2tok() and tok2lex().

The keytoktab.c file has a declaration for tab struct as in figure below and based on this, two tabs are created for keywords and other lexemes. So whenever there is a call to lex2tok() it looks in lexeme tab and return the associated token and if a call is made for key2tok() it looks in keywords tab and return the integer associated with that key word but if the call is to tok2lex() it has to look in both table sequentially.

```
typedef struct tab {

wchar_t * text;
int token;
} tab
```

Figure 4.3 Data structure used to define the lexeme-token combination . Source: own

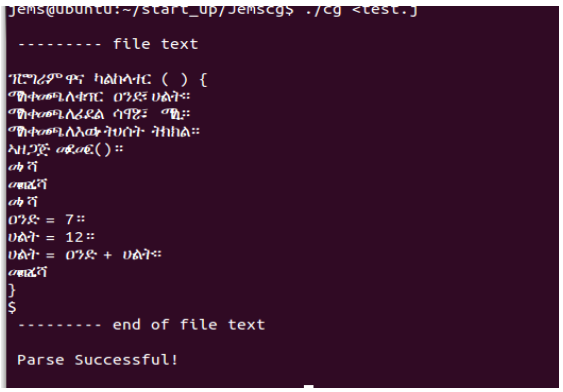


Figure 4.4 Screen capture of a successful compilation. Source: own

4.4. Syntax analyzer

Syntax as defined by Noam Chomsky (Chomsky, 1957) is the study of the principles and processes by which sentences are constructed in particular languages. Syntactic investigation of a given language has as its goal the construction of a grammar that can be viewed as a device of some sort for producing the sentences of the language under analysis.

Syntax analyzer or parser is the part where previously identified tokens are grouped into grammatical phrases that are used by the next phase of the compiler to synthesize the output. This is the part is where any syntax violation will be identified and reported. And those violations are identified by looking into the grammar of the language.

In parser, there is a function for each and every grammar rule of the language and a function to match a terminal. It results in function call if there is a non-terminal or a token matching is done for terminal with the help of match() function. The match() function compares the current token with the expected token and if they are the same it gets the next token by calling the get_token() function defined in the lexer else reports a syntax error and change the value of status indicator variable, is_parse_ok, to zero.

The parser checks if the program is syntactically correct based on the specified grammar of the language. It reports if there is any grammar error and there is an error message based on the error type. Syntax error is when the programmer violates the grammar of the language, for example missing '::' at the end of statement or if there is chains of syntactic units that do not conform to the syntax of the source language.

The parser also make use of functions in keytoktab.c to convert token into lexeme and to get lexeme in wprintf() function. The code listed as figure 4.4 below shows the partial source code used to parse the grammar rule given here.

<stat_part> = ['ውህኒ' <stat_list> 'ውፅጨረሻ']*

<stat_list> = (<statement>'::')*

<statement> = <assign_stat> | <input_stat> | <output_stat> | <if_stat> | <for_stat> | <fun_call> | <return_stat>

<assign_stat> = <variable> '=' <expression>

<variable> = <name>

// <assignment> => <variable> = < expression> is parsed here

```
void assign_stat()
{
    int op=undef, x=undef;
        wchar_t lobuf[10];
        if(DEBUG) printf("\n *** In assign_stat");
        wcscpy(lobuf, get_lexeme());
        op=variable();
        match(assign);
        x=expression();
        assignm(lobuf, lbuf, L"", L"");
        if (x!=op){num_error++;
         is_parse_ok=0;
         wprintf(L"\n *** የዓይነት ዕለመመሳሰል ስህተት");
        }
}
// Identify the statement type based on the initial token and call the function for it
void statement()
{
        if(DEBUG) printf("\n *** In statement");
        switch(lookahead){
        case NAME: assign_stat(); break;
        case output: output_stat(); break;
        case input: input_stat(); break;
        case tif: if_stat(); break;
        case tfor: for_stat(); break;
        case call: func_call(); break;
        case treturn: return_stat(); break;
        default: wprintf(L"\n ስህተት-የአረፍተ ነገሩ አይነት አልተገለጠም");
        }
}
```

```
//<stat_list> => <stat>:: is realized here

void stat_list()

{

        if(DEBUG) printf("\n *** In stat_list");

        statement();

        match(fullstop);

}

// check to see if it is a statment

int is_statment()

{

        if(DEBUG) printf("\n *** In is_statment");

        return(lookahead==tif || lookahead==tfor || lookahead==input || lookahead==output ||
lookahead==NAME || lookahead==call || lookahead==treturn);

}

/* The grammar <stat_part> => begin <stat_list> end  is <stat_list> could be empty so call
is_statment to see */

void stat_part()

{

        if(DEBUG) printf("\n *** In stat_part");

        match(begin);

        while(is_statment(lookahead))stat_list();

        match(end);

}
```

Figure 4.5 code fragment to parse statement. Source: own.

4.5. Semantic analyzer

Semantic analyzer is an implicit part of a compiler with the responsibility of identifying semantic errors. Semantic errors are those which violate the meaning specification of the language, like operations conducted on incompatible types, undeclared variables, double declaration of variable, reference before assignment.

This part uses the operator table to perform its main function, type checking, which defines the legal operator with their respective return type. Using the operator table Jempas performs a type checking for arithmetic operators and for relational operator it does not need the help of the operator table. It checks if the return type is a Boolean or not.

The parser extensively uses the symbol table to see if a variable is declared or if a name is unique. A variable is declared before its first use in statement and added in to the symbol table entry. If it is already exist in symbol table or there is extra character(s) after the closing curl bracket, then the parsing will fail and a proper error message will be given.

4.6. Symbol table

Symbol tables are data structures that are used by compilers to hold information about source program constructs. The information is put into the symbol table when the declaration of an identifier is analyzed. A semantic action gets information from the symbol table when the identifier is subsequently used, for example, as a factor in an expression.

Both front end and back end of a compiler uses the symbol table. The analysis part collects the information for symbol table entry which then used by the synthesis part to generate the target machine code.

Entries in the symbol table contain information about an identifier such as its character string (or lexeme), its type, its position in storage, and any other relevant information. Symbol tables typically need to support multiple declarations of the same identifier within a program. The symbol table in Jempas is design in such a way to facilitate the storage of the basic information about the source program constructs. There are public functions which help other part to access the content of the symbol table. And those functions are declared in symtab.h file which could be included as a user defined header file for any part which will access the symbol table.

The content of the symbol table could be printed from any program which has included the symtab.h header file properly by calling the p_symtab() function. This will print the name, role, type, size and address of each entry in row. If a program wants to add an entry into the symbol table, it should call one of the public functions. The public functions are; *addp_name(wchar_t program_name)* to add the name of the program into the symbol table, role will be set to *ፕሮግራም* and type to *የተገለጠ*, *addf_name(wchar_t function_name)* to add a function name into the entry of the symbol table, role is set to *ፋንክሽን* and type to *የተገለጠ*, *addv_name(wchar_t variable_name)* to add a variable name setting its role and type respectively to *ማስቀመጫ* and *ያልተገለጠ*.

Additional functions to set and get type of a variable when the type clause is realized or if a variable is used in expression and to find name before adding it into the symbol table to avoid double declaration of a name are available in the form of public scope. These functions are used in semantic check part for type checking and error handling. A print out of the symbol table for a test program is given below.

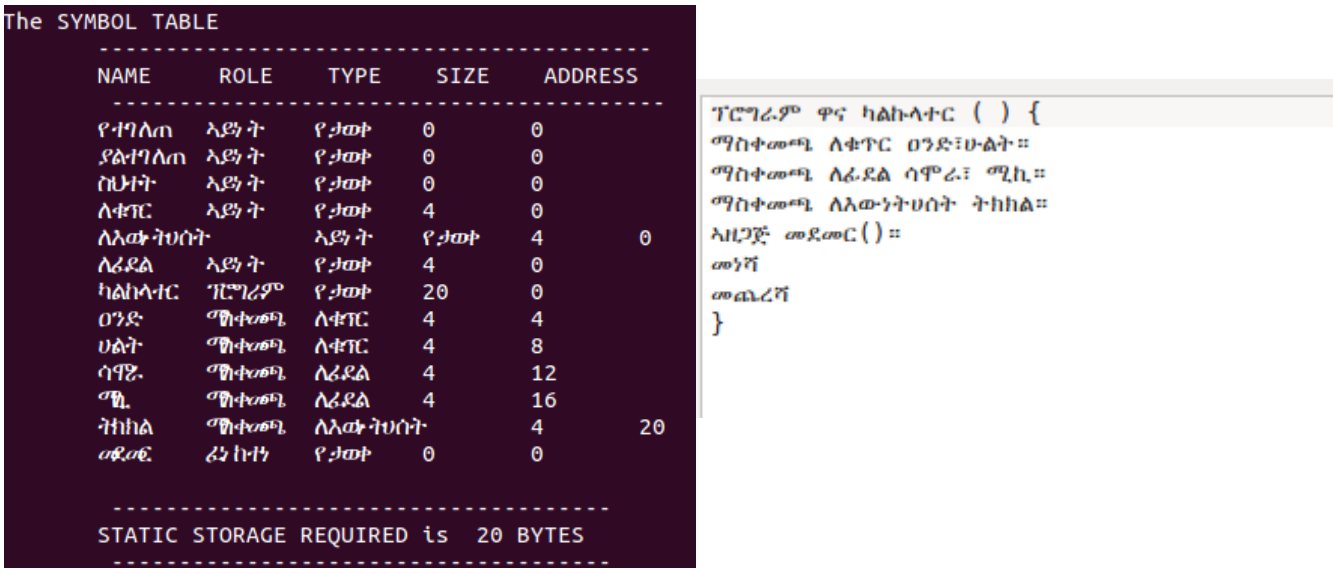


Figure 4.6 Symbol table content (left) and test program (right) Source:own

4.7. Operator table

The operator table is designed to support the type checking function of the compiler. Type checking is one of the error handling in semantic checking and one of the main tasks that a compiler should solve. This table is arranged in optab.h and optab.c with the public like function in optab.h header file.

The optab.c defines a data structure to support the table structure as *static int optab[][Size]* where size is defined in macro as four. Implying the table has four entries in each row, which represent the operator with its argument and the resulting type. A sample code is given in the figure below.

The operator table has a definition for *get_otype(op, arg1, arg2)* which returns the resulting type on realizing an expression, where *op* is a binary operator, *arg1* is the type of the first argument and *arg2* is the type of the second argument. It also has a definition for *p_optab()* the call of which will result on the displaying of the current definition of supported operators and types. These two functions could be imported to any file by including the header file *optab.h.*

```
#define NEVENTS 4
static int optab[][NEVENTS] = {
  '+', integer, integer, integer,
  '-', integer, integer, integer,
  '*', integer, integer, integer,
  '/', integer, integer, integer,
  equal, integer, integer, boolean,
  equal, boolean, boolean, boolean,
  tand, boolean, boolean, boolean,
  tor, boolean, boolean, boolean,
  '$', undef,   undef,   undef
  };
```

Figure 4.7 The operator table data structure

4.8. Intermediate code generator

Intermediate code is a code that is generated by the first pass of a compiler. Rather than translating source code directly from one language to another, compilers first translate it to this more generic and easier to manipulate language and then spit it at the code generator, which creates the finished product. The semantic phase of a compiler first translates parse trees into an intermediate representation (IR), which is independent of the underlying computer architecture, and then generates machine code from the IRs. This makes the task of retargeting the compiler to computer architecture easier to handle.

There are three types of intermediate representation; syntax trees, postfix notation and three-address code (TAC). A syntax tree depicts the natural hierarchical structure of a source program. Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the in which a node appears immediately after its children. Three-address code is a sequence of statements of the general form $x = y\ Op\ z$ where x, y, and z are names, constants, or compiler-generated temporaries; op stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on Boolean-valued data. The reason for the term "three-address code" is that each statement usually contains three addresses, two for the operands and one for the result.

Although there are other intermediate codes specific to the language being implemented like, P-code for Pascal and Byte code for java, the language independent TAC is supported by Jempas. TAC is not only language independent it is also easy to generate, understand and convert it to assembly and machine code.

Jempas produce its final output into two files, *code.tac* and *symtab.stb*. These files could be used in the next phase of a compiler, code generation. Symtab.stb is the symbol table file and code.tac is the TAC file. The simple command used to compile is *gcc –o cg *.c* followed by *./cg <test9.j* but a make file is also provided with command to perform different unit test on each phase and their respective driver.

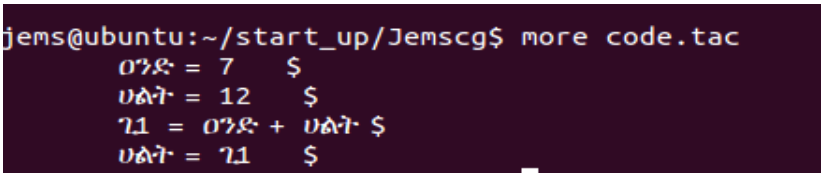


Figure 4.8 The content of code.tac for test9.j on ubuntu screen

5. Results and Discussion

The result of this thesis is Jempas, a compiler that runs on linux, Unix or ubuntu and a grammar for the language it compile. Jempas is totally open source and for academic purpose only, on current version, and any development from academic community, therefore, is welcomed. The source file is well organized, commented and a module driver is included to support unit test for each part of the compiler.

The input file for the Jempas could have any extension provided that the editor is Unicode aware, has support for Amharic input method and the owner would identify it is a Jempas source file. But it will be consistent if every individual could use .j files.

Given a grammatically and semantically correct source code the output of Jempas is a three address code in code.tac file, a symbol table in symtab.stb file and a parse successful notification in the terminal, which makes Jempas useful in teaching the following courses: Compiler Construction, formal language theory, procedural programming, and part of computer architecture. Whereas, if the source code is ill formed syntactically or semantically the output is a message on the type of error, and a notification of a failed parsing.

Since java might not be a good place to start learning programming for beginners, procedural programming like c and Pascal comes in to play in teaching communities. Jempas will play a great role in smoothing the steep learning curve of programming for most of Ethiopian students in the field of technologies.

Since incremental development is employed in the development it is easy to make any kind of development in future. Development in this approach is performed by adding one functionality at a time and testing if it works. Driver for each module is included in the source code file. It was really help full to use this approach of development for that you know at which point your code stop working.

6. Conclusion and recommendation

On this thesis it is clearly shown that there is much to do on area of theoretical computing specifically in relation to Amharic and many of the Unicode only supported languages. As the theoretical computer science is the building block for current development, fellow researchers' origin from Unicode only supported language should give attention to this area of study. It is a successful start in developing amharic programing language and implementing its compiler, and that was the objective of this thesis.

This thesis could be an inspiration for further work in developing Amharic based compilers and programming languages. One can extend the result of this thesis to make a commercial compiler and develop a back end of the Jempas which convert the TAC to machine code or add features to it, like IDE.

It is recommended to use incremental development approach during development of any software, so that creating test scenario will be relatively easy and you will have bug free code at any point in time.

Academic institutions in Ethiopia, by current setup, are focused to teach science using already available tools and methods or do field research focused on existing problem mostly agriculture related. But it will be more valuable and attract young researchers if the academic institutions develop tools tailored to the student. When teaching material and tools are tailored to the student, the student will have a chance to grasp the knowledge as easy as talking to someone with mother tongue. Furthermore, the student will develop the creativity, the problem solving and the logical thinking ability, which in turn will allow the development of new idea, technology and solutions.

Based on the economic, social and political advantages of open source software, developing economy should cooperate and advocate the open source projects. Localization will merely take advanced understanding of the open sourced software and little or no knowledge of programming. Therefore, it is recommended to use open source software and localize them whenever possible rather than violating local, regional or international copy right laws.

7. References

Abay, A. A. 2004. *Compiler construction using flex and bison.* Washington: Walla Walla College

Aho, A.V., Lam, M.S., Sethi, R. & Ullman, J.D. 2007. *Compilers: principles, techniques & tools*(2nd ed.). Boston: Addison Wesley

ASCII, ASA X3.4-1963, American Standards Association, June 17, 1963

Chomsky, N. 1959a. On certain formal properties of grammars. Information and

Control 2: 137–67.

Chomsky, N. 1959b. A note on phrase structure grammars. Information and

Control 2: 393–95.

Chomsky, N. (1956). "Three Models for the Description of Language". *IRE Transactions on Information Theory* **2** (2): 113–123.

Chomsky, N. (1957). *Syntactic Structures*. The Hague: Mouton.

Levelt, W.J.M. (2008). *An introduction to the theory of formal languages and automata.* Amsterdam: John Benjamins Publishing Company.

Meliyu. "AxumLight Features." *AxumLight Features*. Meliyu, June 2011. Web. 07 Feb. 2013

Watt, David A. (2004). *Programming language design concepts.* Chichester: Johan Wiley & Sons, Ltd.

Vanicek, J., Papik, M., Pergl, R., Vanicek T.(2008). *Mathematical foundations of computer science.* Prague: Alfa Publishing.

ISO 7185:1990. Information technology: Programming Languages -- Pascal

ISO/IEC 8859-1:1998. Information technology: 8-bit single-byte coded graphic character sets -- Part 1: Latin alphabet No. 1

The Linux Information Project (2007) Characters. Available at: http://www.linfo.org/character.html (Accessed: 11 June 2012).

The Unicode Consortium (2012) Charts. Available at: http://www.unicode.org/chart.html (Accessed: 11 June 2012).

8. Appendix

8.1. BNF grammar of Jempas

<program> = ’ፕሮግራም’ ’ዋና’ NAME ’(’ ’)’ ’{’ ‹var_part› ‹fun_part› ‹stat_part›’}’

<var_part> = [’ማስቀመጫ’ (<var_decl>)+]*

<var_decl> =[var_dec›(‘፣’var_decl)*]

<Var_dec>=[‹type› ‹NAME> ‘::’]

<type> = ’ለቁጥር’ | ’ለሀረግ’ | ’ለእውነትሀሰት’ | ’ባዶ’

<fun_part> = [<fun_heading><fun_def>]*

<fun_head> = ‘አዘጋጅ’ <name> ‘(’ <formals> ‘)’ [‘፤’ <type>]’::’

<fun_def> = = ‘መነሻ’ <stat_list>‘መጨረሻ’ ‘::’

<formals> = [<var_decl> (‘፣’ <var_decl>)*]

<stat_part> = [‘መነሻ’ <stat_list> ‘መጨረሻ’]*

<stat_list> = (<statement>’።።’)*

<statement> = <assign_stat> | <input_stat> | <output_stat> | <if_stat> | <for_stat> | <fun_call> | <return_stat>

<assign_stat> = <variable> ‘=’ <expression>

<input_stat> = ‘ተቀበል’ ‘(’<name>‘)’

<output_stat> = ‘ስጥ’ ‘(’ <expression> ‘)’

<if_stat> = ‘ከሆነ’ <expression> <stat_list> [‘ካልሆነ’ <stat_list>]

<for_stat> = ‘ከ’ <expression> ‘እስከ’ <expression> <stat_list>

<func_call> = ‘ጥራ’ <name> ‘(’ [<expression> (‘፣’ <expression>)*] ‘)’

<return_stat> = ‘መልስ’ <expression> | ‘መልስ’

<expression> = <simple_expression> [<RELOP> <simple_expression>]

<simple_expression> = <term> (<ADDOP> <term>)*

<term> = <factor> (<MULOP> <factor>)*

<factor> = ‘(’ <expression> ‘)’ | <variable> | <constant> | <func_call>

<variable> = <name>

<constant> = <int_constant> | <boolean_constant> | <string_constant>

<int_constant> = <digit>+

<boolean_constant> = ‘እውነት’ | ’ሀሰት’

<string_constant> = ‘”” <alph_numer>+ ‘””

<alph_numer> = <letter> | <digit> | ‘!’ | ‘?’ | ‘.’

<RELOP> = ‘>=’ | ‘<=’ | ‘==’ | ‘=’ | ‘<’ | ‘>’ | ‘<>’

<ADDOP> = ‘+’ | ‘-’ | ‘ወይም’

<MULOP> = ‘*’ | ‘/’ | ‘እና’

<name> = <letter> (<idchar>)*

<idchar> = <letter> | <digit> | ‘_’

<digit> = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

<letter> =’ሀ’ | ‘ሁ’ | ‘ ሂ’ | ‘ ሃ’ | ‘ ሄ’ | ‘ ህ’ | ‘ ሆ’ | ‘ ለ’ | ‘ ሉ’ | ‘ ሊ’ | ‘ ላ’ | ‘ ሌ’ | ‘

’ | ‘ ሎ’ | ‘ ሐ’ | ‘ ሑ’ | ‘ ሒ’ | ‘ ሓ’ | ‘ ሔ’ | ‘ ሕ’ | ‘ ሖ’ | ‘ መ’ | ‘ ሙ’ | ‘ ሚ’ | ‘ ማ’ | ‘ ሜ’ | ‘ ም’ | ‘ ሞ’ | ‘ ሠ’ | ‘ ሡ’ | ‘ ሢ’ | ‘ ሣ’ | ‘ ሤ’ | ‘ ሥ’ | ‘ ረ’ | ‘ ሩ’ | ‘ ሪ’ | ‘ ራ’ | ‘ ሬ’ | ‘ ር’ | ‘ ሮ’ | ‘ ሰ’ | ‘ ሱ’ | ‘ ሲ’ | ‘ ሳ’ | ‘ ሴ’ | ‘ ስ’ | ‘ ሶ’ | ‘ ሸ’ | ‘ ሹ’ | ‘ ሺ’ | ‘ ሻ’ | ‘ ሼ’ | ‘ ሽ’ | ‘ ሾ’ | ‘ ቀ’ | ‘ ቁ’ | ‘ ቂ’ | ‘ ቃ’ | ‘ ቄ’ | ‘ ቅ’ | ‘ ቆ’ | ‘ በ’ | ‘ ቡ’ | ‘ ቢ’ | ‘ ባ’ | ‘ ቤ’ | ‘ ብ’ | ‘ ቦ’ | ‘ ተ’ | ‘ ቱ’ | ‘ ቲ’ | ‘ ታ’ | ‘ ቴ’ | ‘ት’ | ‘ ቶ’ | ‘ ቸ’ | ‘ ቹ’ | ‘ ቺ’ | ‘ ቻ’ | ‘ ቼ’ | ‘ ች’ | ‘ ቾ’ | ‘ ኀ’ | ‘ ኁ’ | ‘ ኂ’ | ‘ ኃ’ | ‘ ኄ’ | ‘ ኅ’ | ‘ ኆ’ | ‘ ነ’ | ‘ ኑ’ | ‘ ኒ’ | ‘ ና’ | ‘ ኔ’ | ‘ ን’ | ‘ ኖ’ | ‘ ኘ’ | ‘ ኙ’ | ‘ ኚ’ | ‘ ኛ’ | ‘ ኜ’ | ‘ ኝ’ | ‘ ኞ’ | ‘ አ’ | ‘ ኡ’ | ‘ ኢ’ | ‘ ኣ’ | ‘ ኤ’ | ‘ እ’ | ‘ ኦ’ | ‘ ከ’ | ‘ ኩ’ | ‘ ኪ’ | ‘ ካ’ | ‘ ኬ’ | ‘ ክ’ | ‘ ኮ’ | ‘ ኸ’ | ‘ ኹ’ | ‘ ኺ’ | ‘ ኻ’ | ‘ ኼ’ | ‘ ኽ’ | ‘ ኾ’ | ‘ ወ’ | ‘ ዉ’ | ‘ ዊ’ | ‘ ዋ’ | ‘ ዌ’ | ‘ ው’ | ‘ ዎ’ | ‘ ዐ’ | ‘ ዑ’ | ‘ ዒ’ | ‘ ዓ’ | ‘ ዔ’ | ‘ ዕ’ | ‘ ዖ’ | ‘ ዘ’ | ‘ ዙ’ | ‘ ዚ’ | ‘ ዛ’ | ‘ ዜ’ | ‘ ዝ’ | ‘ ዞ’ | ‘ ዟ’ | ‘ ዠ’ | ‘ ዡ’ | ‘ ዢ’ | ‘ ዣ’ | ‘ ዤ’ | ‘ ዥ’ | ‘ ዦ’ | ‘ የ’ | ‘ ዩ’ | ‘ ዪ’ | ‘ ያ’ | ‘ ዬ’ | ‘ ይ’ | ‘ ዮ’ | ‘ ደ’ | ‘ ዱ’ | ‘ ዲ’ | ‘ ዳ’ | ‘ ዴ’ | ‘ ድ’ | ‘ ዶ’ | ‘ ጀ’ | ‘ ጁ’ | ‘ ጂ’ | ‘ ጃ’ | ‘ ጄ’ | ‘ ጅ’ | ‘ ጆ’ | ‘ ገ’ | ‘ ጉ’ | ‘ ጊ’ | ‘ ጋ’ | ‘ ጌ’ | ‘

’ | ‘ ጎ’ | ‘ ጠ’ | ‘ ጡ’ | ‘ ጢ’ | ‘ ጣ’ | ‘ ጤ’ | ‘ ጥ’ | ‘ ጦ’ | ‘ ጨ’ | ‘ ጩ’ | ‘ ጪ’ | ‘ ጫ’ | ‘ ጬ’ | ‘ ጭ’ | ‘ ጮ’ | ‘ ጯ’ | ‘ ጰ’ | ‘ ጱ’ | ‘ ጲ’ | ‘ ጳ’ | ‘ ጴ’ | ‘ ጵ’ | ‘ ጶ’ | ‘ ጷ’ | ‘ ጸ’ | ‘ ጹ’ | ‘ ጺ’ | ‘ ጻ’ | ‘ ጼ’ | ‘ ጽ’ | ‘ ጾ’ | ‘ ፀ’ | ‘ ፁ’ | ‘ ፂ’ | ‘ ፃ’ | ‘ ፄ’ | ‘ ፅ’ | ‘ ፆ’ | ‘ ፇ’ | ‘ ፈ’ | ‘ ፉ’ | ‘ ፊ’ | ‘ ፋ’ | ‘ ፌ’ | ‘ ፍ’ | ‘ ፎ’ | ‘ ፐ’ | ‘ ፑ’ | ‘ ፒ’ | ‘ ፓ’ | ‘ ፔ’ | ‘ ፕ’ | ‘ ፖ’ | ‘ ቐ’ | ‘ ቑ’ | ‘ ቒ’ | ‘ ቓ’ | ‘ ቔ’ | ‘ ቕ’ | ‘ ቖ’ | ‘ ቘ’ | ‘ ቚ’ | ‘ ቛ’ | ‘ ቜ’

8.2 Keywords of the language.

✓	E1 8D 95 E1 88 AE E1 8C 8D E1 88 AB E1 88 9D	ፕሮግራም	program
✓	E1 8B 8B E1 8A 93	ዋና	main
✓	E1 88 9B E1 88 B5 E1 89 80 E1 88 98 E1 8C AB	ማስቀመጫ	var
✓	E1 88 88 E1 89 81 E1 8C A5 E1 88 AD	ለቁጥር	int
✓	E1 88 88 E1 88 83 E1 88 A8 E1 8C 8D	ለሃረግ	char
✓	E1 88 88 E1 8A A5 E1 8B 8D E1 8A 90 E1 89 B5 E1 88 80 E1 88 B0 E1 89 B5	ለእውነትሀሰት	boolean
✓	E1 89 A3 E1 8B B6	ባዶ	void
✓	E1 8A A0 E1 8B 98 E1 8C 8B E1 8C 85	አዘጋጅ	define
✓	E1 88 98 E1 8A 90 E1 88 BB	መነሻ	begin
✓	E1 88 98 E1 8C A8 E1 88 A8 E1 88 BB	መጨረሻ	end
✓	E1 89 B0 E1 89 80 E1 89 A0 E1 88 8D	ተቀበል	input
✓	E1 88 B5 E1 8C A5	ስጥ	output
✓	E1 8A A8 E1 88 86 E1 8A 90	ከሆነ	if
✓	E1 8A AB E1 88 8D E1 88 86 E1 8A 90	ካልሆነ	else
✓	E1 8A A8	ከ	from
✓	E1 8A A5 E1 88 B5 E1 8A A8	እስከ	to
✓	E1 8C A5 E1 88 AB	ጥራ	call
✓	E1 8A A5 E1 8B 8D E1 8A 90 E1 89 B5	እውነት	true
✓	E1 88 80 E1 88 B0 E1 89 B5	ሀሰት	false
✓	E1 8B 88 E1 8B AD E1 88 9D	ወይም	or
✓	E1 8A A5 E1 8A 93	እና	and
✓	E1 88 98 E1 88 8D E1 88 B5	መልስ	return

8.3. Utf-8 representation of the characters in the language

Base symbol	1	2	3	4	5	6	7
ሀ	E1 88 80	E1 88 81	E1 88 82	E1 88 83	E1 88 84	E1 88 85	E1 88 86
ለ	E1 88 88	E1 88 89	E1 88 8a	E1 88 8b	E1 88 8c	E1 88 8d	E1 88 8e
ሐ	E1 88 90	E1 88 91	E1 88 92	E1 88 93	E1 88 94	E1 88 95	E1 88 96
መ	E1 88 98	E1 88 99	E1 88 9a	E1 88 9b	E1 88 9c	E1 88 9d	E1 88 9e
ሠ	E1 88 a0	E1 88 a1	E1 88 a2	E1 88 a3	E1 88 a4	E1 88 a5	E1 88 a6
ረ	E1 88 a8	E1 88 a9	E1 88 aa	E1 88 ab	E1 88 ac	E1 88 ad	E1 88 ae
ሰ	E1 88 b0	E1 88 b1	E1 88 b2	E1 88 b3	E1 88 b4	E1 88 b5	E1 88 b6
ሸ	E1 88 b8	E1 88 b9	E1 88 ba	E1 88 bb	E1 88 bc	E1 88 bd	E1 88 be
ቀ	E1 89 80	E1 89 81	E1 89 82	E1 89 83	E1 89 84	E1 89 85	E1 89 86
በ	E1 89 a0	E1 89 a1	E1 89 a2	E1 89 a3	E1 89 a4	E1 89 a5	E1 89 a6
ተ	E1 89 b0	E1 89 b1	E1 89 b2	E1 89 b3	E1 89 b4	E1 89 b5	E1 89 b6
ቸ	E1 89 b8	E1 89 b9	E1 89 ba	E1 89 bb	E1 89 bc	E1 89 bd	E1 89 be
ኀ	E1 8a 80	E1 8a 81	E1 8a 82	E1 8a 83	E1 8a 84	E1 8a 85	E1 8a 86
ነ	E1 8a 90	E1 8a 91	E1 8a 92	E1 8a 93	E1 8a 94	E1 8a 95	E1 8a 96
ኘ	E1 8a 98	E1 8a 99	E1 8a 9a	E1 8a 9b	E1 8a 9c	E1 8a 9d	E1 8a 9e
አ	E1 8a a0	E1 8a a1	E1 8a a2	E1 8a a3	E1 8a a4	E1 8a a5	E1 8a a6
ከ	E1 8a a8	E1 8a a9	E1 8a aa	E1 8a ab	E1 8a ac	E1 8a ad	E1 8a ae
ኸ	E1 8a b8	E1 8a b9	E1 8a ba	E1 8a bb	E1 8a bc	E1 8a bd	E1 8a be
ወ	E1 8b 88	E1 8b 89	E1 8b 8a	E1 8b 8b	E1 8b 8c	E1 8b 8d	E1 8b 8e
ዐ	E1 8b 90	E1 8b 91	E1 8b 92	E1 8b 93	E1 8b 94	E1 8b 95	E1 8b 96
ዘ	E1 8b 98	E1 8b 99	E1 8b 9a	E1 8b 9b	E1 8b 9c	E1 8b 9d	E1 8b 9e
ዠ	E1 8b a0	E1 8b a1	E1 8b a2	E1 8b a3	E1 8b a4	E1 8b a5	E1 8b a6
የ	E1 8b a8	E1 8b a9	E1 8b aa	E1 8b ab	E1 8b ac	E1 8b ad	E1 8b ae
ደ	E1 8b b0	E1 8b b1	E1 8b b2	E1 8b b3	E1 8b b4	E1 8b b5	E1 8b b6
ጀ	E1 8c 80	E1 8c 81	E1 8c 82	E1 8c 83	E1 8c 84	E1 8c 85	E1 8c 86

ᦀ	E1 8c 88	E1 8c 89	E1 8c 8a	E1 8c 8b	E1 8c 8c	E1 8c 8d	E1 8c 8e
ᦠ	E1 8c a0	E1 8c a1	E1 8c a2	E1 8c a3	E1 8c a4	E1 8c a5	E1 8c a6
ᦨᦱ	E1 8c a8	E1 8c a9	E1 8c aa	E1 8c ab	E1 8c ac	E1 8c ad	E1 8c ae
ᦰ	E1 8c b0	E1 8c b1	E1 8c b2	E1 8c b3	E1 8c b4	E1 8c b5	E1 8c b6
ᦸ	E1 8c b8	E1 8c b9	E1 8c ba	E1 8c bb	E1 8c bc	E1 8c bd	E1 8c be
ᧀ	E1 8d 80	E1 8d 81	E1 8d 82	E1 8d 83	E1 8d 84	E1 8d 85	E1 8d 86
ᧈᧁ	E1 8d 88	E1 8d 89	E1 8d 8a	E1 8d 8b	E1 8d 8c	E1 8d 8d	E1 8d 8e
᧐	E1 8d 90	E1 8d 91	E1 8d 92	E1 8d 93	E1 8d 94	E1 8d 95	E1 8d 96
᧞	E1 89 90	E1 89 91	E1 89 92	E1 89 93	E1 89 94	E1 89 95	E1 89 96
᧡	E1 89 a8	E1 89 a9	E1 89 aa	E1 89 ab	E1 89 ac	E1 89 ad	E1 89 ae