A New Compiler: Code Conversion at Assembly Level

Ritu Sindhu, Neha Gehlot, Indu Malik

Abstract: Ever switched programming languages? If yes, you know how difficult it is to learn the syntax and get familiar with new language. But what if we write the code in our preferred language and it can run as any other language's code. The thing is, whatever we write ultimately gets converted to 0's and 1's, the only difference is how these 0's and 1's is shown to our machine. We may need different languages, but what if the code with the syntax of one language, runs reasonably well as if it was written with syntax of some other language. This is where a compiler comes in[1].

The aim of this paper is to develop a compiler which could create a new code for another language, based on the machine code developed by other languages. This compiler solves two problems Syntax issue and Universal Compiler.

Keyword: whatever different languages, languages.

I. INTRODUCTION:

A compiler interprets and/or compiles a program written during a appropriate language into identical target language through variety of stages.
Starting with recognition of token through target code generation offer a basis for communication interface between a user and a processor in important quantity of your time[1]. Various elements of the compiler are as follows:
Source code: It is the high-level code that we write in IDE of a particular language.
Assembly code: It is the, high level code converted to a basic low-level code. It contains mixture of machine code and high-level code.
Machine code: This is the code which machine finally sees.
Compiler: It converts high-level language into assembly code and then the assembler converts assembly code into machine code.

Revised Manuscript Received on January 22, 2020.

Prof. (Dr.) Ritu Sindhu, School of Computing Sciences and Engineering, Galgotias University, Uttar Pradesh, India.
E-mail:ritu.sindhu2628@gmail.com
Ms. Neha Gehlot, Department of Computer Science and Engineering, SGT University, Gurugram, Haryana, India.
E-mail:neha_fet@sgtuniversity.org
Ms. Indu Malik, School of Computing Sciences and Engineering, Galgotias University, Uttar Pradesh, India.
E-mail:ritu.sindhu2628@gmail.com

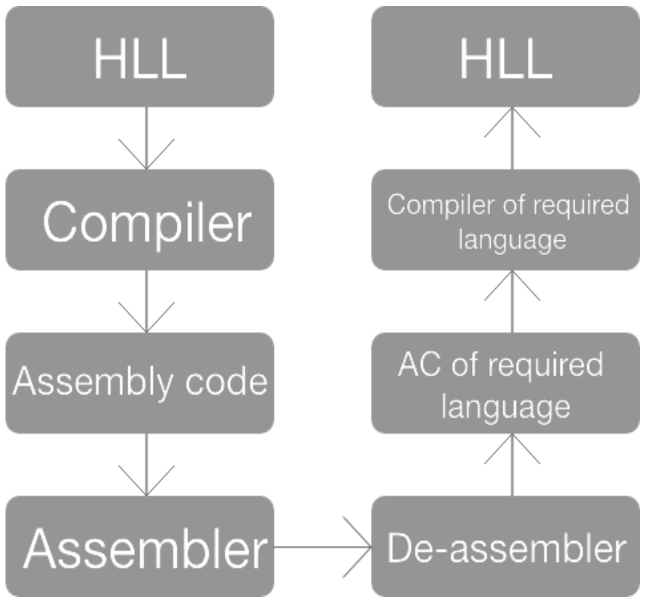
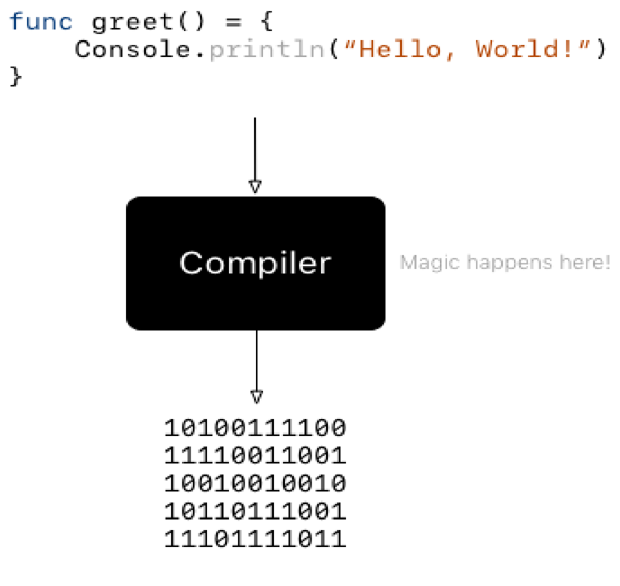


Figure 1: Block Diagram of Layman language of working of suggested compiler

The idea of making a language to be able to run as any other code written in different language, is that the compiler will be a combination of all compilers available for every language and it will check which language syntax the user is using and accordingly generates the assembly code. This assembly code is then sent to a de-assembler of the language in which we want our code to execute. This de-assembler returns the assembly code in required language which is then decompiled to achieve the code conversion from one language to another.



```
func greet() = {
    Console.println("Hello, World!")
}
```

```
10100111100
11110011001
10010010010
10110111001
11101111011
```

So, the most challenging part is deigning this de-assembler. De-assembler has to solve two problems firstly it has to understand the assembly code of one language. Secondly it must be capable of converting code of source language to other languages [4].

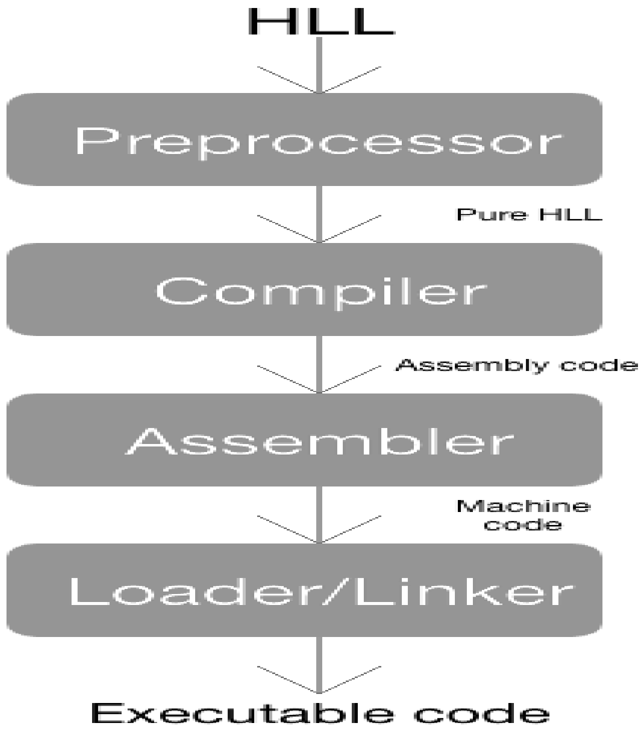


Figure 2: Overview of working of compiler

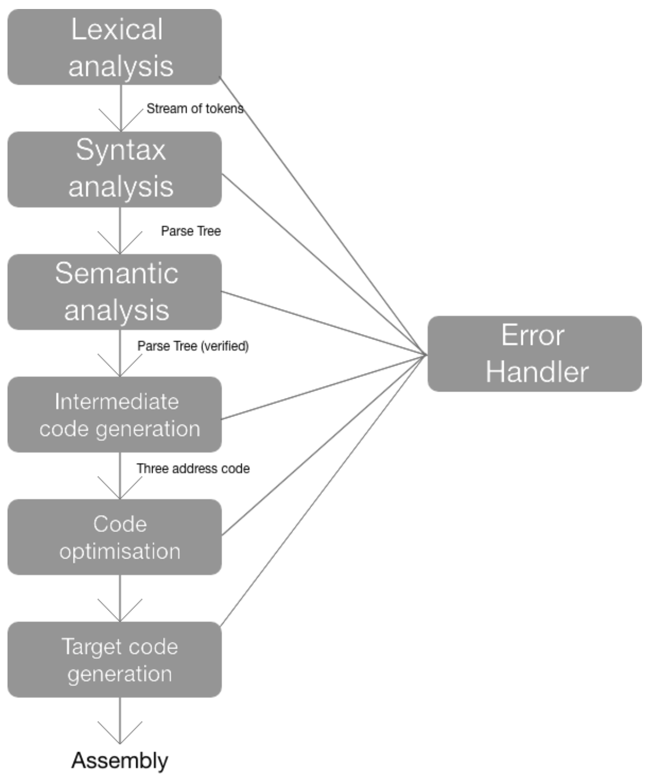


Figure 3: Structure of compiler

So the question is how the language code can be converted to another language code?

The thing is if we try to convert syntax of one language to another language, it would consume lot of memory with checks and loops[3].

So we take the code into assembly level and then recompile it using reverse of compilation or de-compiler.

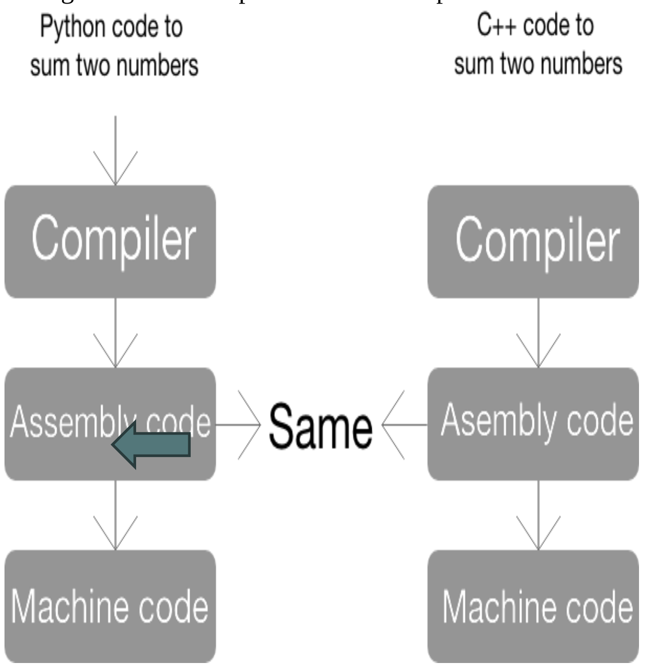
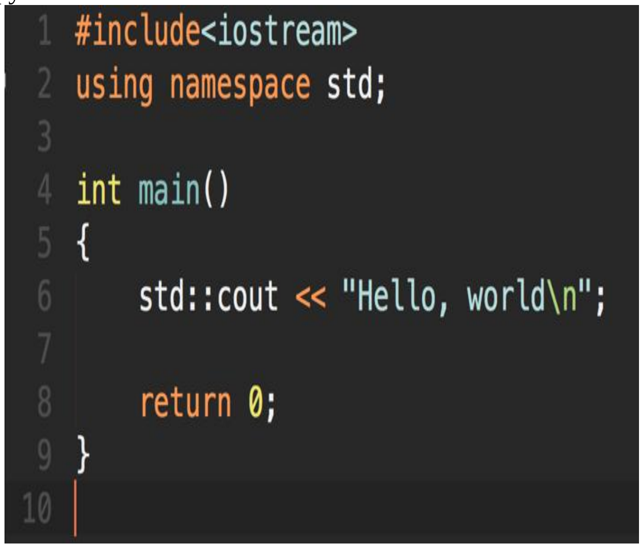


Figure 4: Conversion of code

So, with the proper understanding of compiler we can use the assembly code of python and feed it to De-assembler of C++ to set the equivalent C++ code of program written in python.



```
#include<iostream>
using namespace std;

int main()
{
    std::cout << "Hello, world\n";

    return 0;
}

```

Figure 5 : Core concept of working of code conversion

Suppose we have a C++ program as shown in above fig. We want our compiler to give a python equivalent code for this. So, we start as follows.

Step1: Preprocessor- It removes all the whitespaces from our C++ code and converts in into pureHLL.

Step2: Lexical analysis-(From here the compilation begins) It converts the pure HLL into stream of token that is it breaks up the code into smallest possible units.

Step3: Syntax analysis-This phase of compiler checks whether the syntax of our C program is correct or not and accordingly forwards a parse tree of the code with instructions to the next phase.

Step4: Semantic analysis- This phase checks whether the declaration and statements of a program has clear meaning and are consistent in a way which control statements and data types are supposed to be used. After verifying a semantically verified parse tree is forwarded.

Step5: Intermediate code generation-The semantically verified parse tree is converted to a linear representation (e.g. postfix notation). The intermediate code tends to machine independent. To evaluate this linear representation a three-address code is generated and is forwarded for code optimization.

Step6: Code optimization-The work of code optimization is to make the three address code compact and robust.

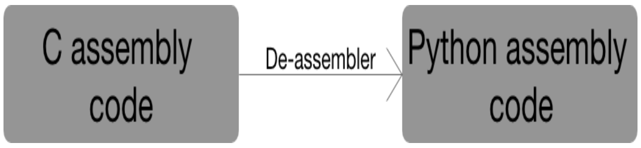
Step7: Target code optimization-It's the final test of our C++ code after which assembly code is generated and then passed to the assembler[14].

II. RESULT ANALYSIS:

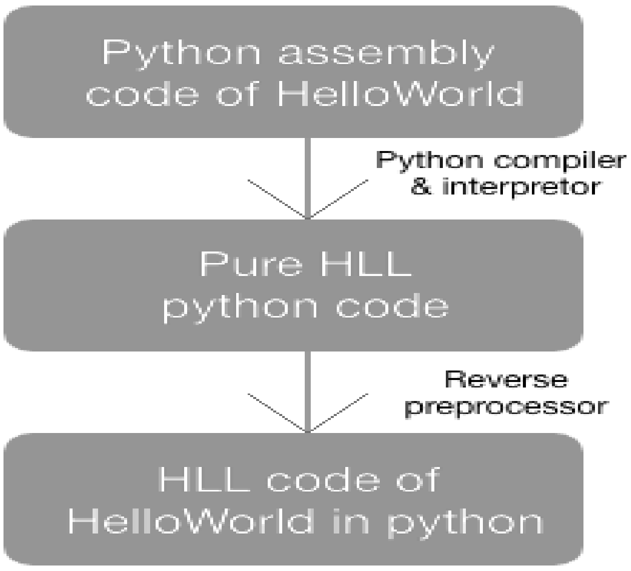
Up till now we have normal compiler deign. After converted to assembly our code and result of compilation looks like.

```
File Type: COFF OBJECT
main:
 0000000000000000: 48 83 EC 28              sub
rsp,28h
  0000000000000016: C3                      ret
__local_stdio_printf_options:
  0000000000000000:  48  8D  05  00  00  00          lea
rax,[?_OptionsStorage@?1??__local_stdio_printf_options@
@9@9]
                  00
  0000000000000007: C3                      ret
_vfprintf_l:0000000000000000: 4C 89 4C 24 20        mov
qword ptr [rsp+20h],r9
 0000000000000014: 48 83 EC 38          sub      rsp,38h
  0000000000000018: E8 00 00 00 00      call
__local_stdio_printf_options
  0000000000000039: E8 00 00 00 00      call
__stdio_common_vfprintf
  000000000000003E: 48 83 C4 38         add      rsp,38h
  0000000000000042: C3                  ret
printf:
  0000000000000000: 48 89 4C 24 08      mov      qword ptr
[rsp+8],rcx
  0000000000000027: E8 00 00 00 00      call
__acrt_iob_func
  000000000000002C: 4C 8B 4C 24 28      mov      r9,qwordptr
[rsp+28h]
  000000000000003C: E8 00 00 00 00      call     _vfprintf_l
  0000000000000041: 89 44 24 20         movdwordptr
[rsp+20h],eax
  000000000000004E: 8B 44 24 20         moveax,dwordptr
[rsp+20h]
  0000000000000056: C3
```

The tricky thing begins from here. Since assembler is platform dependent, so it will convert our C++ assembly code to different machine code for different platform. So if we want to go for code conversion we must revert back our C assembly code to a assembly equivalent of python[15].



This phase conversion is where the core design idea of this paper lies. Working at a hardware level, if we get an exact conversion, then we can revert back the process of compilation using python compiler to get our desired result.


Python assembly code of HelloWorld
Python compiler & interpretor
Pure HLL python code
Reverse preprocessor
HLL code of HelloWorld in python

III. CHALLENGES:

- The phase conversion of assembly code of our language to another language's assembly code is the most important problem to tackle down.
- The designed compiler must learn to identify what language is being used so that we can program a hybrid language and get its equivalent code in any other language, at advance level.
- Generating high level language code from assembly code, it's the reverse process of compiling so I call it de-compiling.
- Initially I thought of developing a ML model that would classify code, this could serve as a potential challenge in developing such a thing and integrating it with ML.

IV. CONCLUSION:

I was really fascinated by this thought of mine, a concept of universal compiler which solves a great problem where we need different compilers depending upon programming language and on the other hand it's also capable of generating code in different languages vanishing the language compatibility in the entire technical industry.

With this compiler being implemented a programmer can rid of learning new languages, simple feed his code to this compiler and just get the code in desired language. Programming language will never be a barrier in implementing things and solving real world problem which is at the end of the day every one's main motive.
Further concluding, my entire design process of this thing will be divided into 3 steps firstly I would design the basic structure of compiler which will be the mixture of all the compilers present in the today's date. Secondly the main part would be to develop a translator which would translate the assembly code of language to the assembly code of another language, this will work at the assembly level. Last part would be de-compiler which would generate HLL code form assembly code. The most challenging phase is the second phase which I have already discussed in the challenges section of this paper.

REFERENCES:

1. Aho, Alfred V., Hop croft, J. E., and Ullman, Jeffrey D. [1974]. The Design andAnalysis of Computer Algorithms.Addision Wesley, Reading, MA
2. Ball, T., Larus, J.: Optimally profiling and tracing programs. ACM Transactions on Programming Languages and Systems 16(3), 1319–1360 (1994).
3. Ball, T., Larus, J.: Efficient path profiling. In: Proc. 29th Annual Intl. Symp. on Microarchitecture (December 1996).
4. Berkeley Unified Parallel C (UPC) Project.
5. Berstein, D., Rodeh, M.: Global Instruction Scheduling for Superscalar Machines. In: Proc. of SIGPLAN 1991 Conference on Programming Language Design and Implementation (1991).
6. Calder, B., Feller, P., Eustance, A.: Value Profiling. In: Proc. 30th Annual Intl. Symp. on Microarchitecture (December 1997.
7. https://stackoverflow.com/questions/47948234/assembly-code-of-a-hello-world-program-in-c
8. https://nptel.ac.in/courses/106/105/106105190/
9. Johnstone, A., Scott, E.: Modelling GLL parser implementations. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 42–61. Springer, Heidelberg (2011).
10. Salomon, D.J., Cormack, G.V.: Scannerless NSLR(1) Parsing of Programming Languages. In: Programming Language Design and Implementation, PLDI 1989, pp. 170–178 (1989).
11. Visser, E.: Scannerless Generalized-LR Parsing. Technical report, University of Amsterdam (1997)
12. https://stackoverflow.com/questions/840321/how-can-i-see-the-assembly-code-for-a-c-program.
13. Ijirt.org.
14. www.codon.uk.org.uk.
15. ijarece.org.

AUTHORS PROFILE

Dr. Ritu Sindhu pursued her Master of Technology from Banasthali University, Rajasthan, India. She did Her Ph.D from Banasthali University,Rajasthan, India.. She is currently working as a Professor, School of Computer Science and Engineering,Galgotias University, Greater Noida, India. She has published 40 research papers in various reputed National and International Journals. Her teaching experience is 14 Years.

Ms.Neha Gehlot pursued her Master of Technology from ITM University, Gurugram India. She is pursuing her Ph.D from SGT University , Gurugram India. She is currently working as a Assistant Professor, in SGT University, Gurugram India. She has published 15 research papers in various reputed National and International Journals. Her teaching experience is 5 Years.

Ms. ndu is working as an Assistant Professor in Galgotias University. She have completed M.Tech in Computer Science & Engineering from Banasthali Vidyapith. And have worked on lidar data during my M.Tech project.