Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 3, Issue. 10, October 2014, pg.850 – 852

RESEARCH ARTICLE

COMPILER BASIC DESIGN AND CONSTRUCTION

Mahak Jain[1], Nidhi Sehrawat[2], Neha Munsi[3]
[1]Computer Science and Engineering, Maharshi Dayanand University, India
[2]Computer Science and Engineering, Maharshi Dayanand University, India
[3]Computer Science and Engineering, Maharshi Dayanand University, India
[1] mahakjain140993@gmail.com; [2] nidhisherawat90@gmail.com; [3] nehamunsi@gmail.com

Abstract- Compiler construction is a widely used software engineering exercise, and hence this paper presents a compiler system for adaptive computing. The final result of this paper is to provide a general knowledge about compiler design and its implementation. In order to develop effective compi-lation techniques, it is important to understand the common characteristics of the programs during compi-lation. Although this paper concentrates on the implementation of a compiler, an outline that builds upon the compiler is also presented.

Keywords: Lexer-Parser, Competence development, Compi-lation, Lexical analysis, Syntactic analysis reconfigurable hardware

I. INTRODUCTION

Computer programs are formulated in a programming language and specify classes of computing processes. This paper provides an introduction to the new learning method for compiler construction and designing, which aims at learning compiler design. Computers however, interpret sequences of instructions, but not the program texts. Therefore, the program text needs to be interpreted into a suitable instruction sequence before it is processed by a computer. This paper is aimed at people interested in new learning approaches for understanding basis in computer science engineering. Compilers and operating systems constitute the basic interfaces between a programmer and the machine. Basically, Compiler is a program which converts high level programming language into low level programming language or is converts source code into machine code.

The core compiler reads a program described in a high-level programming language. The compiler then analyses the program, partitions it into hardware and software, and then generates data paths for the reconfi-gurable hardware. It focuses on the basic relationships between languages and machines. Therefore, the relationships ease the inevitable transitions to new hardware and programming languages. In parallel, the software part is instrumented with functions for configuring and exchanging with the reconfi-gurable hardware.

The term compi-lation denotes the conversion of an algorithm expressed in a human-oriented source language to an algorithm expressed in a hardware-oriented target language. Also consider Conventional programs give priority to knowledge in which competency is flexible and adaptable and cannot be reduced to an algorithm.

Programming languages are the tools used to construct formal descriptions consists of finite computations (algorithms), in which each computation further consists of operations that transform a given initial state into the final state. In the context of factual information that can consist of, for example, a definition, a theorem, a hypothesis, a rule, or an algorithm. We shall be concerned with the engineering of compilers.

II. LEXICAL

The lexical syntax (token structure), which is processed by the lexer and the phrase syntax, which is processed by the parser. The lexical syntax is usually a regular language, whose alphabet consists of the individual characters of the source code text. The phrase syntax is usually a context-free language, whose alphabet consists of the tokens produced by the lexer.

In computer science, lexical analysis is the process of converting a sequence of characters into a sequence of tokens, i.e. meaningful character strings. A program or function that performs lexical analysis is called a lexical analyzer, lexer, tokenizer, or scanner, though "scanner" is also used for the first stage of a lexer.

III. PARSER

Within computational linguistics the term is used to refer to the formal analysis by a computer of a sentence or other string of words into its constituents, resulting in a parse tree showing their syntactic relation to each other, which may also contain semantic and other information.

The term has slightly different meanings in different branches of linguistics and computer science. In order to parse natural language data, researchers must first agree on the grammar to be used. The choice of syntax is affected by both linguistic and computational concerns; traditional sentence parsing is often performed as a method of understanding the exact meaning of a sentence, sometimes with the aid of devices such as sentence diagrams. It usually emphasizes the importance of grammatical divisions such as subject and predicate.

Parsing or syntactic analysis is the process of analysing a string of symbols, either in natural language or in computer languages.

IV. LEXER-PARSER PROCESSING

While using a lexical scanner and a parser together, the parser is the higher level routine. These generators are a form of domain-specific language, taking in a lexical specification – generally regular expressions with some mark-up and outputting a lexer. The lexer then scans through the input recognizing tokens. Automatically generated lexer may lack flexibility, and thus may require some manual modification or a completely manually written lexer.

The next stage is parsing or syntactic analysis, which is checking that the tokens form an allowable expression. This is usually done with reference to a context-free grammar which recursively defines components that can make up an expression and the order in which they must appear. However, not all rules defining programming languages can be expressed by context-free grammars alone, for example type validity and proper declaration of identifiers. These rules can be formally expressed with attribute grammars. This can be done in essentially two ways:

A. **Top-down parsing**- Top-down parsing can be viewed as an attempt to find left-most derivations of an input-stream by searching for parse trees using a top-down expansion of the given formal grammar rules.

B. **Bottom-up parsing** - A parser can start with the input and attempt to rewrite it to the start symbol. Intuitively, the parser attempts to locate the most basic elements, then the elements containing these, and so on. LR parsers are examples of bottom-up parsers. Another term used for this type of parser is Shift-Reduce parsing.

V. OPTIMAL STORAGE MANAGEMENT

The important goals are the most economical use of memory and the simplicity of access functions to individual objects and hence said Optimal.

In Static Storage Management, if the compiler can provide fixed addresses for all objects at the time the program is translated. This can be done by or the condition is fulfilled for languages like FORTRAN and BASIC, and for the objects lying on the outermost contour of an ALGOL 60 or Pascal program.

While if the storage for the elements of an array with dynamic bounds is managed separately, then the condition can be forced to hold. That is particularly interesting when we have additional information that certain procedures are not recursive, as in recursive programs the storage is being determined from analysis of the procedure calls.

VI. CONCLUSION

Besides covering basic compilation issues, the course yields an implemented compiler that can serve as a test bed for coursework implementation for compiler. We described an improved approach for a compiler which partitions a high-level language program. Further research will actually quantify the advantages in relation to the current system. The implementation and source language is Scheme, and the target language is assembly code.

REFERENCES

[1] GerhardGoosInstitutProgrammstrukturen und DatenorganisationFakultat fur Informatik

[2] University at KarlsruheD-76128 KarlsruheGermanyemail: ggoos@ipd.info.uni-karlsruhe.de

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

[4] http://www.esa.informatik.tu-darmstadt.de/twiki/pub/Staff/AndreasKochPublications/2001_ERSA01.pdf

[5] http://www.ijsrp.org/research-paper-0413/ijsrp-p16108.pdf