"A Study on System Programming and Compiler Construction and Its Phases"

Anu Batra

Assistant Professor, Dronacharya Institute of Management & Technology, Kurukshetra, Pin Code – 136118, Haryana (India)

Abstract – System programming is the activity of computer programming system software. The primary distinguishing characteristic of systems programming when compared to application programming is that application programming aims to produce software which provides services to the user e.g. word processor, whereas systems programming aims to produce software which provides services to the computer hardware e.g. disk defragmenter. It requires a greater degree of hardware awareness.

----------♦----------

INTRODUCTION

Systems programming is sufficiently different from application programming that programmers tend to specialize in one or the other. In system programming, often limited programming facilities are available. The use of automatic garbage collection is not common and debugging is sometimes hard to do. The runtime library, if available at all, is usually far less powerful, and does less error checking. Because of those limitations, monitoring and logging are often used; operating systems may have extremely elaborate logging subsystems.

Implementing certain parts in operating system and networking requires systems programming, for example implementing Paging (Virtual Memory) or a device driver for an operating system.

Originally systems programmers invariably wrote in assembly language. Experiments with hardware support in high level languages in the late 1960s led to such languages as PL/S,BLISS, BCPL and extended ALGOL for Burroughs large systems. Forth also have applications as a systems language. In the 1980s, C became ubiquitous, aided by the growth of Unix. More recently C++ has seen some use, for instance a subset of it is used in the I/O Kit drivers of Mac OS X.

The following attributes characterize systems programming:

- The programmer will make assumptions about the hardware and other properties of the system that the program runs on, and will often exploit those properties, for example by using an algorithm that is known to be efficient when used with specific hardware.
- Usually a low-level programming language or programming language dialect is used that:
- can operate in resource-constrained environments
- is very efficient and has little runtime overhead
- has a small runtime library, or none at all
- allows for direct and "raw" control over memory access and control flow
- lets the programmer write parts of the program directly in assembly language
- Often systems programs cannot be run in a debugger. Running the program in asimulated environment can sometimes be used to reduce this problem.

SYSTEM PROGRAMMING LANGUAGE

A System programming language is usually used to mean "a language for system programming": that is, a language designed for writing system software as distinct from application software**.**

System software is computer software designed to operate and control the computer hardware and to provide a

platform for running application software, and includes such things as operating systems, utility software, device drivers, compilers, and linkers**.**

FEATURES

In contrast with application languages, system programming languages typically offer more-direct access to the physical hardware of the machine: an archetypical system programming language in this sense was BCPL. System programming languages often lack built in input/output facilities because a system-software project usually develops its own input/output or builds on basic monitor I/O or screen management services facilities. The distinction between languages for system programming and applications programming became blurred with widespread popularity of PL/I, C and Pascal**.**

The earliest system software was written in assembly language for reasons including efficiency of object code, compilation time, and ease of debugging. Application languages such as FORTRAN were used for system programming, although they usually still required some routines to be written in assembly language.

Mid-level languages "have much of the syntax and facilities of a higher level language, but also provide direct access in the language as well as providing assembly language to machine features." One of the earliest of these mid-level programming languages was PL360, which had the general syntax of ALGOL 60, but whose statements directly manipulated CPU registers and memory. Other languages in this category are MOL-360 and PL/S.

As an example, a typical PL360 statement is R9 := R8 and R7 shall 8 or R6, signifying that registers 8 and 7 should be together, the result shifted left 8 bits, the result of that oriented with the contents of register 6, and the result placed into register 9**.**

While PL360 is at the semantic level of assembly language, another kind of system programming language operates at a higher semantic level, but has specific extensions designed to make the language suitable for system programming. An early example of this kind of language is LRLTRAN which extended Fortran with features for character and bit manipulation, pointers, and directly-addressed jump tables.

Subsequently, languages such as C were developed, where the combination of features was sufficient to write system software, and a compiler could be developed that generated efficient object programs on modest hardware. Such a language generally omits features that cannot be implemented efficiently, and adds a small number of machine-dependent features needed to access specific hardware capabilities; inline assembly code, such as C's asm statement, is often used for this purpose. Although many such languages were developed, C and C++ are the ones that have survived**.**

System Programming Language (SPL) is also the name of a specific language on the HP 3000 computer series, used for its operating system HP Multi-Programming Executive, and other parts of its system software.

Compiler construction is an area of computer science that deals with the theory and practice of developing programming languages and their associated compilers. The theoretical portion is primarily concerned with syntax, grammar and semantics of programming languages. One could say that this gives this particular area of computer science a strong tie with linguistics. Some courses on compiler construction will include a simplified grammar of a spoken language that can be used to form a valid sentence for the purposes of providing students with an analogy to help them understand how grammar works for programming languages. The practical portion covers actual implementation of compilers for languages. Students will typically end up writing the front end of a compiler for a simplistic teaching language, such as Micro**.**

LEXICAL ANALYSIS

The first phase of compilation is lexical analysis of the source code. This phase involves grouping the characters into lexemes. Lexemes belong to *token classes* such as "integer", "identifier", or "whitespace". A *token* of the form <token-class, attribute-value> is produced for each lexeme. Lexical analysis is also called *scanning*.

SYNTAX ANALYSIS

The second phase of constructing a compiler is syntax analysis. The output of lexical analyzer is used to create a representation which shows the grammatical structure of the tokens. Syntax analysis is also called *parsing.*

Compilers and operating systems constitute the basic interfaces between a programmer and the machine for which he is developing software. In this book we are concerned with the construction of the former. Our intent is to provide the reader with a firm theoretical basis for compiler construction and sound engineering principles for selecting alternate methods, implementing them, and integrating them into a reliable, economically viable product. The emphasis is upon a clean decomposition employing modules that can be re-used for many compilers, separation of concerns to facilitate team

programming, and exibility to accommodate hardware and system constraints. A reader should be able to understand the questions he must ask when designing a compiler for language X on machine Y, what trades are possible and what performance might be obtained. He should not feel that any part of the design rests on whim; each decision must be based upon specific, identifiable characteristics of the source and target languages or upon design goals of the compiler.

The vast majority of computer professionals will never write a compiler. Nevertheless, study of compiler technology provides important benefits for almost everyone in the field. It focuses attention on the basic relationships between languages and machines. Understanding of these relationships eases the inevitable transitions to new hardware and programming languages and improves a person's ability to make appropriate in design and implementation. It illustrates application of software engineering techniques to the solution of a significant problem. The problem is understandable to most users of computers, and involves both combinatorial and data processing aspects.

Many of the techniques used to construct a compiler are useful in a wide variety of applications involving symbolic data. In particular, every man-machine interface constitutes a form of programming language and the handling of input involves these techniques. We believe that software tools will be used increasingly to support many aspects of compiler construction.

The details of this discussion are only interesting to those who must construct such tools; the general outlines must be known to all who use them. We also realize that construction of compilers by hand will remain an important alternative, and thus we have presented manual methods even for those situations where tool use is recommended.

Virtually every problem in compiler construction has a vast number of possible solutions. We have restricted our discussion to the methods that are most useful today, and make no attempt to give a comprehensive survey. Thus, for example, we treat only the LL and LR parsing techniques and provide references to the literature for other approaches. Because we do not constantly remind the reader that alternative solutions are available, we may sometimes appear overly dogmatic although that is not our intent.

A decomposition of any problem identifies both tasks and data structures. For example, we discussed the analysis and synthesis tasks. We mentioned that the analyzer converted the source program into an abstract representation and that the synthesizer obtained information from this abstract representation to guide its construction of the target algorithm. Thus we are led to recognize a major data object, which we call the structure tree in addition to the analysis and synthesis tasks.

We define one module for each task and each data structure identified during the decomposition. A module is specified by an interface that defines the objects and actions it makes available, and the global data and operations it uses. It is implemented by a collection of procedures accessing a common data structure that embodies the state of the module. Modules fall into a spectrum with single procedures at one end and simple data objects at the other. Four points on this spectrum are important for our purposes.

Procedure: An abstraction of a single "memory less" action i.e. an action with no internal state. It may be invoked with parameters, and its effect depends only upon the parameter values. Example A procedure to calculate the square root of a real value.

Package: An abstraction of a collection of actions related by a common internal state.

The declaration of a package is also its instantiation, and hence only one instance is possible. Example - The analysis or structure tree module of a compiler.

Abstract data type: An abstraction of a data object on which a number of actions can be performed. Declaration is separate from instantiation, and hence many instances may exist. Example: A stack abstraction providing the operations push, pop, top, etc.

Variable: An abstraction of a data object on which exactly two operations, fetch and store, can be performed. Example : An integer variable in most programming languages.

Abstract data types can be implemented via packages: The package defines a data type to represent the desired object, and procedures for all operations on the object. Objects are then instantiated separately. When an operation is invoked, the particular object to which it should be applied is passed as a parameter to the operation procedure.

REFERENCES:

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools.
- Michael Wolfe. High-Performance Compilers for Parallel Computing. ISBN 978-0-8053-2730-4.

- Aho, Alfred V.; Lam, Monica S.; Sethi, Ravi; Ullman, Jeffrey D. (2007), Compilers: Principles, Techniques, & Tools (2nd ed.), Pearson, ISBN 978-81-317-2101-8.
- Rissen, J. P., Heliard, J. C., Ichbiah, J. D., and Cousot, P. The system implementation language LIS, reference manual. Technical Report 4549 E/EN, CII Honeywell-Bull, Louveciennes, France.
- Robertson, E. L. Code generation and storage allocation for machines with span dependent instructions. ACM Transactions on Programming Languages and Systems, 1(1):71{83.Rohrich, J. Automatic construction of error correcting parsers. Technical Report Internert Bericht 8, University at Karlsruhe.
- Rohrich, J. Methods for the automatic construction of error correcting parsers. ActaInformatica, 13(2):115.
- Rosen, S. . Programming and Systems and Languages. Mc Grawhill.
- Rosenfeld, J. L., editor . Information Processing 74. North-Holland, Amsterdam, NL.
- Rosenkrantz, D. J. and Stearns, R. E. Properties of deterministic top-down grammars. Information and Control, 17:226.
- Ross, D. T. . The AED free storage package. Communications of the ACM, 10(8):481- 492.
- Rutishauser, H. Automatische Rechenplanfertigung bei Programm-gesteuerten
- Rechenmaschinen. Mitteilungen aus dem Institut four Angewandte Mathematik der ETHZourich.
- Sale, Arthur H. J. The classification of FORTRAN statements. Computer Journal, 14:10.