Reviews of Books and Papers in the Computer Field

DONALD L. EPLEY, Reviews Editor
D. W. FIFE, A. I. RUBIN, R. A. SHORT, H. S. STONE
Assistant Reviews Editors

Please address your comments and suggestions to the Reviews Editor: Donald L. Epley,
Department of Electrical Engineering, University of Iowa, Iowa City, Iowa 52240

A. COMPILERS


Since BNF (context-free) grammars have been found so useful in describing the syntax of programming languages, a large number of parsing algorithms for context-free grammars have been developed for use in compilers and compiler-writing systems. A significant class of these algorithms have the following properties.

1) The algorithm works only on a subset S of the context-free grammars.
2) Given a grammar from S, a construction algorithm compiles a parser which is tailored to that particular grammar.
3) The speed of the compiled parser is Cn, where n is the length of the string being parsed and C1 is independent of the size of the grammar or n.
4) The space required by the compiled parser is Cm+Cn, where C1 is independent of n or the grammar but C2 depends on the grammar.

The crucial parameters in evaluating an algorithm of this type are C1, C2, and C3. C3 is usually negligible. If S is too small, one may have difficulty getting a grammar for the programming language which the parser will accept. If C1 is too large, the parser may be too slow, and if C2 grows too quickly with the size of the grammar, then the compiler parser may not fit in core for some large grammars. The algorithm described in this paper is of the type having these four properties, so we will evaluate it by the above criteria.

Its main advantage seems to be a small C1, making it quite fast. This is obtained by using a transition matrix as a switching table which lets one determine from the top symbol of the stack (denoting a row of the table) and the next symbol of the program to be processed (represented by a column of the table) whether a reduction should be made, or whether the incoming symbol should be pushed onto the stack. This technique has been used in other compilers, and this paper is an attempt to formalize and automate it.

The set of grammars which are accepted by the algorithm is a subset of the operator grammars which satisfies certain complex restrictions. The author mentions that it is a subset of (1, 1) bounded context grammars, but gives no other indication of its relationship to the sets of grammars acceptable to other fast parsers. Most grammars for full programming languages are not operator grammars, and many are not even (1, 1) bounded context, so the set of acceptable grammars for this algorithm is probably smaller than would be preferable. This difficulty can be overcome by changing the grammar (without changing the language it generates), but I suspect that it is rather difficult and time consuming to maneuver a large grammar into exactly the form necessary to satisfy the restrictions of this algorithm.

The space C1 is taken up mainly by the transition matrix. Its two dimensions are the number of terminals in the grammar and the number of "starred" nonterminals added to the grammar (the construction algorithm first converts its operator grammar into a special form, adding some nonterminals). This second number seems to be about as large as the number of original nonterminals in the grammar. The matrix size for an AIGOL compiler is 45 by 100, so the space used is large, but not enough to exceed the memory of most large computers, except possibly for extremely large grammars, such as PL/1.

In conclusion, the algorithm is quite fast, and if one has either a large memory or a small grammar and is willing to spend the effort needed to produce an acceptable grammar for his language, then the algorithm could be quite useful.

I would have preferred to see more comparison of this algorithm with other similar algorithms (such as Floyd's operator precedence algorithm) by the three criteria I have mentioned. I found the mathematical part of the paper complicated and hard to follow, but the description of the implementation and the rest of the paper are clearly written and the examples well chosen.

At the end of the paper, a section is devoted to the use of parsing algorithms for operations other than the usual syntax checking. While it has little to do with the rest of the paper, it is interesting.

Line (4.8) contains an error; it should read

\[ U_1^*U_2^* \cdots U_{n-1}^*U_1U_2T_1 \cdots T_n. \]

JAY EARLEY
Dept. of Computer Science
University of California
Berkeley, Calif.

B. TIME-SHARED SYSTEM SCHEDULING


This narrative comprises a rather subjective and limited summary of queueing analyses that are applicable to the design of computer scheduling algorithms. The queueing disciplines are all treated
as priority disciplines, and are categorized according to priority
dependence on running time, system state, external factors, or wait-
ing time. A single-server system is assumed for most of the discussion,
but a section on multiple-server systems is included at the end of the
paper.

The summary would seem to be of most value to someone enter-
ing queuing theoretic work with a commitment to computer applica-
tions. The bibliography is an acceptable entry to the literature,
where the mathematical developments are given. The discussion
points out a sufficient number of alternatives to show the potential
of this area for interesting and challenging problems.

D. W. Fife
Lexington, Mass.

C. MULTIPROGRAMMING

R68-48 A Multiprogramming Monitor for Small Machines—G. D.

This paper describes a monitor system for a small display com-
puter, attached as a satellite to the Berkeley SDS-940 time-sharing
system. It describes some modifications made to the interrupt-
handling hardware of the small machine, a DEC PDP-8. It also gives
details of a set of calls to the monitor, which allow multiprogramming
of tasks in the satellite.

The author's competence is demonstrated by the way he has
tackled three important features of the system. First, he has added
a useful extension to the PDP-8 interrupt system: this includes an
arm/disarm feature, such as can be found on a number of recent SDS
designs, and a flag scanner. Second, he has kept the size of the monitor
down to 300 words, a remarkable achievement considering the weak
instruction set of the PDP-8. Third, he has recognized that a satel-
lite display computer must be capable of executing the user's real-
time programs. A number of similar systems have been produced
which do not possess this capability, because the satellite can only
refresh the display and process interrupts.

The author appears to have devoted little attention to a fourth,
and equally important, aspect of the system, namely, the provision of
a simple and convenient language for writing real-time graphical
programs. Instead, these must be written in PDP-8 assembly lan-
guage. The author has provided a number of monitor calls, such as
FORK, WAIT, and CONTINUE, which may be imbedded in such pro-
grams; in this way the user can multiprogram the various input and
output tasks. The paper does not offer any convincing reason why
I/O should be multiprogrammed in this way. An example is given,
of identifying objects on the screen by pointing at them; applying
multiprogramming techniques, this becomes a program of quite
needless complexity, with four FORKS and six separate multipro-
grammed tasks. This may have been a poor choice of example, or
the author may have programmed it in this way in order to demon-
strate the use of multiprogramming. One is left with the suspicion
that it could have been greatly simplified by not using multipro-
gramming techniques, and that the same probably applies to the
majority of graphical programs.

The paper is well written, and presents a very clear account of
the system.

William M. Newman
University of Utah
Salt Lake City, Utah

R68-49 Virtual Memory Processes and Sharing in MULTICS—R. C.

This paper gives a description of the "virtual machine" of MULTICS,
the multiple-user computer system conceived at M.I.T. and imple-
mented on the General Electric Model 645 time-sharing computer.
The virtual machine is the machine which is seen by the system
"user," i.e., the application programmer working at the assembly
language level. The MULTICS virtual machine has a number of salient
features which distinguish it from conventional machines.

1) The virtual machine provides a very large (2^38 words) "virtual
memory" for each of potentially many concurrent processes, where
a process is defined to be the activity carried out by the system in
executing a given user's program. A virtual memory address has two
components: a) a segment number, identifying one of as many as 2^14
segments per process, and b) a word number, identifying a word within
a segment. Segments may consist of executable instructions ("pro-
cedure segments") or of data ("data segments"). The large size of the
virtual memory relieves the user of the burden of preplanning the
transfer of data and instructions between physical storage devices,
and makes users' programs independent of the nature of these stor-
age devices. The "two-dimensional" virtual memory simplifies the
writing of programs involving multiple data arrays of dynamically
varying size; the program has, in effect, two edges to grow on instead
of the usual one.

2) Procedure segments and data segments may be shared among
a number of concurrent processes. This prevents core storage from
becoming cluttered with multiple copies of the same routines and
files.

3) A process may refer to procedure segments and data segments
dynamically, i.e., the decision to use a given segment may be deferred
until the actual running of the program. In this manner, virtual
memory is allocated only to those segments which are actually re-
quired.

4) Any segments may be recompiled and substituted for its
original version without change to the segments which refer to it
(provided the change does not affect the external characteristics of
the segment). This feature simplifies the development of segment
libraries which are shared by many users.

The foregoing features imply certain capabilities in the system
hardware and software which are not found in conventional systems.
For example, the large virtual memory implies a hardware "paging"
mechanism in which pages of instructions and data are transferred
between core storage and some form of secondary storage as required
by the active processes; and an address translation mechanism which
translates virtual memory addresses into actual core addresses while
the program is being executed (dynamic program relocation). The
sharing of procedure segments among unrelated processes imposes
certain restrictions on the way such segments are written, e.g., they
cannot modify themselves. The dynamic referencing and substitution
of segments implies that such references be held in symbolic form,
and that a translation from the symbolic form to the virtual memory
address form be carried out the first time the reference is encountered
in a process.

This paper is concerned mainly with explaining how the foregoing
capabilities are accomplished in MULTICS, particularly the capability
for the dynamic translation of symbolic addresses. Almost no attempt
is made to justify these capabilities, presumably on the assumption
that the reader is already convinced of their value. (For readers not
so convinced, a number of references are given. In particular, an
earlier work of Dennis gives some useful arguments for the concept
of a two-component virtual memory.) The paper is carefully and
accurately written.

W. C. McGee
IBM Corporation
Palo Alto, Calif.

1 J. B. Dennis, "Segmentation and the design of multiprogrammed computer