Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

21

F. L. Bauer · F. L. De Remer · M. Griffiths
U. Hill · J. J. Horning · C. H. A. Koster
W. M. McKeeman · P. C. Poole · W. M. Waite

Compiler Construction

An Advanced Course

Edited by F. L. Bauer and J. Eickel

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

21

F. L. Bauer · F. L. De Remer · M. Griffiths
U. Hill · J. J. Horning · C. H. A. Koster
W. M. McKeeman · P. C. Poole · W. M. Waite

Compiler Construction

An Advanced Course

Edited by F. L. Bauer and J. Eickel

Springer-Verlag
Berlin · Heidelberg · New York 1974

Prof. Dr. F. L. Bauer
Prof. Dr. J. Eickel
Mathematisches Institut
der TU München
8 München 2
Arcisstraße 21
BRD

AMS Subject Classifications (1970): 00A10, 68-02, 69-02
CR Subject Classifications (1974): 4.12

ISBN 3-540-06958-5 Springer-Verlag Berlin · Heidelberg · New York
ISBN 0-387-06958-5 Springer-Verlag New York · Heidelberg · Berlin

Offsetdruck: Julius Beltz, Hemsbach/Bergstr.

Contents

CHAPTER 1.: INTRODUCTION

CHAPTER 2.: ANALYSIS

CHAPTER 5.: ENGINEERING A COMPILER

CHAPTER 6.: APPENDIX

CHAPTER 1.A

COMPILER CONSTRUCTION

W. M. McKeeman
The University of California at
Santa Cruz
U. S. A.

"If PL/I is the Fatal Disease,
then perhaps Algol-68 is
Capital Punishment".
An Anonymous Compiler Writer

1. DEFINITIONS

1.1. SOURCE AND TARGET LANGUAGES

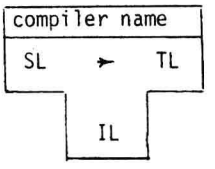
A compiler is a program, written in an implementation language, accepting text in a source language and producing text in a target language. Language description languages are used to define all of these languages and themselves as well. The source language is an algorithmic language to be used by programmers. The target language is suitable for execution by some particular computer.

If the source and target languages are reasonably simple, and well matched to each other, the compiler can be short and easy to implement (See Section 1.A.2 of these notes). The more complex the requirements become, the more elaborate the compiler must be and, the more elaborate the compiler, the higher the payoff in applying the techniques of structured programming.

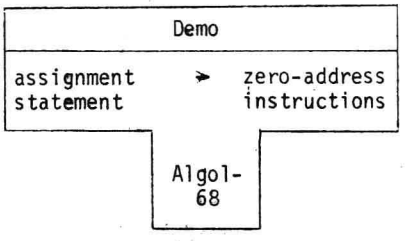
1.2. IMPLEMENTATION LANGUAGES

Compilers can, and have, been written in almost every programming language, but the use of structured programming techniques is dependent upon the implementation language being able to express structure. There are some existing languages which were explicitly designed for the task of compiler writing (FSL [Feldman 66], XPL [McKeeman 70], CDL [Koster 71b], and some for structuring (Pascal [Wirth 71], Algol 68 [van Wijngaarden 68]). The criterion for choosing an implementation language is quite straight forward: it should minimize the implementation effort and maximize the quality of the compiler. Lacking explicit knowledge of this kind, the compiler writer is advised to seek a language as close as possible to the ones mentioned above. The number, quality and availability of such languages is generally on the increase. It may be advantageous to write a compiler to run on a different machine than the target text will run on if better tools can thus be used (especially common for very small target machines). In any case, we shall simply assume an appropriate implementation language is available.

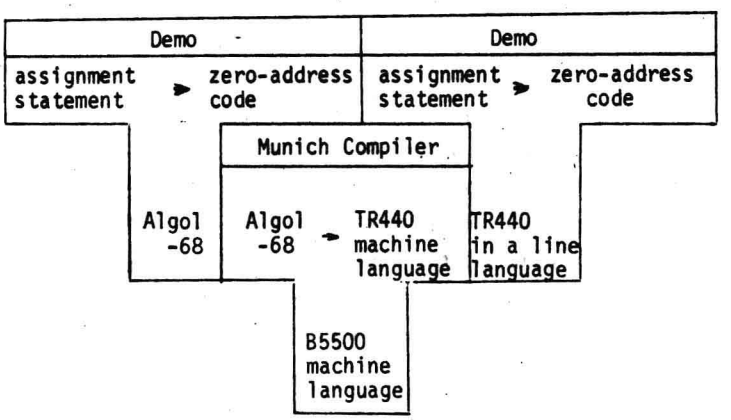
Since there are so many languages involved, and thus so many translations, we need a notation to keep the interactions straight. A given translator has three main languages (SL, TL, IL above) which are objects of the prepositions from, to and in respectively. A T diagram of the form



gives all three [Bratman 61]. If the compiler in Section 1.A.2 (below) is called Demo, then it can be described by the diagram



Now a compiler written in Algol-68 is of no use unless there is also a running compiler for Algol-68 available. Suppose it is on the Burroughs B5500. Then if we apply it to the T above, we will get a new T as follows:



where the arms of the middle T must match the tails of the Ts to the left and right. Complicated, multistage, multilanguage, multimachine translation processes can be described by appropriate cascades of such T diagrams [McKeeman 70 pp. 16-18].

1.3 Language Defining Languages

Language defining languages are almost always based on grammars (see Chapter 2 of these notes) but frequently have additional features designed to define the target text (i.e., translation defining languages). Thus the distinction between language definition and implementation language has not always been very clear. There was a tradition at one point of time to define a programming language as "what the compiler would translate" but this turned out to be of no value to the user who was not prepared to explore the idiosyncracies of a compiler to be able to write programs. The problem then has been to define languages without leaning on the compiler itself.

The ultimate solution is a definitional mechanism both clear enough for human reference and usable as input to a translator writing system which automatically creates the compiler.

2. Recursive Descent Compilation

2.1 Introduction

It is the intent of the following pages to give a concrete example of translation and also to review a particular, rather simple, rather successful, translation technique. The example, a translation of assignment statements to an assembly language for a stack machine, is trivial by contemporary standards but serves to elucidate the process. We can, in fact, present the entire translator as a whole without becoming mired in either detail or side issues. For the example, the source text

A = - A + 5 * B / (B-1)

will be translated to the following zero-address target text
LIT A LIT A LOAD NEG LIT 5 LIT B LOAD MUL LIT B LOAD LIT 1 NEG ADD DIV ADD STORE
which closely approximates the instructions of the Burroughs B5500 computer [Organik 71] (See also Chapters 3.A and 3.E of these notes). The target text is executed, step by step, in Figure 2.1. Although the meaning of the target text is probably obvious to the reader, we will take a few words to describe it. There is an evaluation stack into which values can be pushed. Furthermore, the top values (hence the last ones pushed into the stack) are available to be used in computations. The LIT instruction pushes one value onto the stack. That value is either an address (e.g., the address of the variable A in LIT A) or a constant (e.g. the value 5 in LIT 5).

The LOAD instruction assumes the top value on the stack is an address. The address is removed from the stack and the value found in the indicated cell in memory is pushed onto the stack in its place. The STORE instruction must be supplied with two items at the stack top. One is the address of a cell in memory. The other is a value to be stored into the indicated cell (the address is below the value in the stack). After the STORE instruction has completed its action, both address and value are removed from the stack. The remainder of the instructions are arithmetic operations. NEG changes the sign of the top value on the stack and leaves it where it found it. ADD, MUL and DIV operate on the two top elements on the stack, removing