Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

38

P. Branquart · J.-P. Cardinael · J. Lewi
J.-P. Delescaille · M. Vanbegin

An Optimized Translation Process and Its Application to ALGOL 68

Springer-Verlag
Berlin · Heidelberg · New York 1976

Authors
Paul Branquart
Jean-Pierre Cardinael*
Johan Lewi**
Jean-Paul Delescaille
Michael Vanbegin

MBLE Research Laboratory
Avenue Em. van Becelaere 2
1170 Brussels/Belgium

* Present address: Caisse Générale d'Epargne et de Retraite, Brussels, Belgium

** Present address: Katholieke Universiteit Leuven, Applied Mathematics and Programming Division, Leuven, Belgium

Library of Congress Cataloging in Publication Data
Main entry under title:

An Optimized translation process and its application to ALGOL 68.

(Lecture notes in computer science ; 38)
Bibliography: p.
Includes index.
1. ALGOL (Computer program language)
2. Compiling (Electronic computers) I. Branquart, Paul, 1937- II. Series.
QA76.73.A24067 001.6'424 75-45092

AMS Subject Classifications (1970): 68-02, 68A05, 90-04
CR Subject Classifications (1974): 4.1, 4.12

ISBN 3-540-07545-3 Springer-Verlag Berlin · Heidelberg · New York
ISBN 0-387-07545-3 Springer-Verlag New York · Heidelberg · Berlin

Printed in Germany
Offsetdruck: Julius Beltz, Hemsbach/Bergstr.

FOREWORD

In the late sixties, the definition of ALGOL 68 [1], for a long time called ALGOL X, reached some stability. It is at that period (1967) our team started the project of writing a compiler for that language. We had two goals in mind :

(1) to make significant research in the field of compiler methodology,

(2) to point out the special difficulties encountered in the design of the compiler and thus possibly influence the definition of the language.

This book is concerned with the first goal only ; ALGOL 68 should be considered a support to explain and develop compiling principles and techniques.

The whole book is directly based on the actual compiler we have written for the Electrologica-X8 computer ; this compiler has been operational since early 1973. Since May 1975, it is available on the "BS-computer", the Philips prototype developed by MBLE and which is at the origin of the UNIDATA 7720. In fact, the X8 has been microprogrammed on the BS [22] ; it is worthwhile to mention that microprogramming did not introduce any significant loss in efficiency.

The book does not require a very deep knowledge of ALGOL 68 except in some special cases described here for the sake of completeness only. The reading of some general description of the language as provided by [17] is however assumed.

Acknowledgments

We should like to express our thanks to Mrs Micheline Mispelon for her excellent typing of the manuscript and to Mr Claude Semaille for his careful drawing of the figures.

SUMMARY

The book describes a translation process which generates efficient code while remaining machine independent. The process starts from the output stream of the syntactic analyzer.

(1) Code optimization is based on a mechanism controlling a number of static properties and allowing to make long range previsions. This permits to minimize the dynamic (run-time) actions, replacing them by static (compile-time) ones whenever possible. In particular, much attention is paid on the minimization of run-time copies of values, of run-time memory management and of dynamic checks.

(2) Machine independency is improved by translating the programs into intermediate code before producing machine code. In addition to being machine independent, intermediate code instructions are self-contained modules which can be translated into machine code independently, which improves modularity. Only trivial local optimizations are needed at the interface between intermediate code instructions when machine code is produced.

The description of the translation process is made in three parts :

-PART I defines the general principles on which the process is based. It is made as readable as possible for an uninitiated reader.

-PART II enters the details of translation into intermediate code : particular problems created by all ALGOL 68 language constructions and their interface are solved.

-PART III shows the principles of the translation of the intermediate code into machine code ; these principles are presented in a completely machine independent way.

CONTENTS

PART I : GENERAL PRINCIPLES

0. INTRODUCTION

A programming language is defined by means of a *semantics* and a *syntax*.

- the *semantics* defines the meaning of the programs of the language. It is based on a number of *primitive functions (actions)* having parameters, delivering a result and/or having some side-effects, and on a number of *composition rules* by which the result of a function may be used as the parameter of another function.
- the *syntax* provides means for program representations. It defines a structure of programs, reflecting both the primitive functions and the composition rules of the semantics.

A *compiler* translates programs written in a given *source language* into programs written in *an object language* and having the same meaning. Ultimately the object language is the machine code. Generally, the transformation is performed in two steps at least conceptually separated : the *syntactic analysis* and the *translation proper*.

0.1 BASIC CONCEPTS

The *syntactic analysis* is a program transformation by which the structure of the source program is made explicit. We can distinguish three parts in the syntactic analysis, namely :

- the *lexical analysis* by which atoms of information semantically significant in the source language are detected,
- the *context-free analysis* by which the primitive functions of the source language and their composition rules are made explicit, and
- the *declaration handling* by which the declared objects are connected to their declaration.

Conceptually, the output of the syntactic analysis has the form of a tree in which :

- the terminal nodes are the atoms delivered by the lexical analyzer. These atoms may represent values (value denotations, identifiers) or they may just be source language syntactic separators or key-words,
- nonterminal nodes represent functions (actions) the parameters of which are the values resulting from the subjacent nodes ; in turn, these functions may deliver a value as their result, and
- the initial node is obviously the syntactic unit "particular program".

The translation proper produces machine code. Elementary functions of, and values handled by *machine codes* are much more primitive than primitive functions of high level languages and their parameters. The translation process has to decompose the source functions and source values. Machine instructions are executed as indepen-

dent modules : the interface between them is determined by the sequence in which they are elaborated and by the storage allocation scheme on which the program they constitute is based. More concretely, the result of each instruction is stored in a memory cell and it can be used by another instruction in which the access (address) of the same memory cell is specified.

Roughly speaking, machine code generation for a given program is based on the following informations :
- the program tree resulting from the syntactic analysis,
- the semantics of the source functions as defined by the source language, and
- the semantics of the machine instructions as defined by the hardware.

The main task of the compiler reduces to decompose source functions into equivalent sequences of machine instructions. Obviously, a storage allocation scheme must first be designed in order to be able to take the composition rules of the source language into account.

It is not required to produce machine code in one step ; our translation scheme first produces an intermediate form of programs called *intermediate code* (IC). Among other things, this permits to remain machine independent during a more significant part of the translation process and hence to increase the compiler portability. We propose an intermediate code consisting of the same primitive functions as the source language, but provided with explicit parameters making it possible, these functions to be considered separate self-contained modules. As it is the case for the machine code, these modules are elaborated sequentially except when explicit breaks of sequence appear. The composition rules of the source language are taken into account through the sequential elaboration of the modules and the strategy of storage allocation. In this respect, as opposed to the source language dealing with abstract instances of values, the intermediate code deals with stored values characterized by the static properties corresponding both to the abstract instances of values [1] (mode ...) and to the memory locations where the values are stored (access ...). It is those properties which are used as the parameters of the intermediate code (object) instructions (ICI) ; more precisely, the parameters of an ICI consist of one set of static (compile-time) properties for each parameter of the corresponding source function and one set for the result of this function.

Coming back to our translation scheme, we can say that intermediate code generation for a given program is based on the following information :
- the program tree resulting from the syntactic analysis,
- the semantics of the source functions, and
- the storage allocation scheme.

We see that the semantics of machine instructions has disappeared, only the storage allocation can be influenced by the hardware. In fact, we only make two hypotheses at the level of the intermediate code :

- the memory is an uninterrupted sequence of addressable units,
- there exists an indirect addressing mechanism.

Machine independent optimizations are performed at the level of the intermediate code generation. In particular
- run-time copies of values,
- run-time memory management, and
- dynamic checks

are minimized up to a great extent.

Moreover, precautions are taken in order to allow to retrieve machine dependent optimizations in a further step ; such optimizations take care of :
- register allocation and
- possible hardware literal and/or display addressing.

Now, machine code generation can be based on the following :
- the intermediate code form of the programs,
- the semantics of the source functions, and
- the semantics of the machine code.

Note that each intermediate code instruction can be translated independently into machine code which improves the compiler modularity. This translation mainly consists in decomposing source functions and data into machine instructions and words (bytes) respectively. Only local optimizations (peephole [16]) at the interface between ICI's will still be needed to get the final machine code program.

Gathering information to be able to translate a program efficiently and automatically requires a non trivial static (compile-time) information management. The method explained in this book has many similarities with the one described by Knuth [6] , although it has been developed independently. We explain it using Knuth's terminology.

Attributes are static properties attached to the tree nodes ; there are *synthetized* and *inherited* attributes.

In our system, the *synthetized attributes* of a node are the static properties (mode, access ...) of the value attached to the node, i.e. the value of a terminal construction (denotation, identifier) or the value resulting from a function (nonterminal node).

These synthetized attributes are deduced from each other in a bottom-up way. For a terminal node, they are obtained from the terminal construction itself (and from its declaration in case of a declared object). For nonterminal nodes, they are calculated by the process of *static elaboration.*

The *static elaboration* of a function is the process by which the static properties of the result of the function are derived from the static properties of its parameters (i.e. the synthetized attributes of the subjacent nodes) and according to the code generated for the translation of the function.

Again, in our system, *inherited attributes* of a node are attributes which are trans-

mitted in the tree in a top-down way along a path leading from the initial node to the current node.

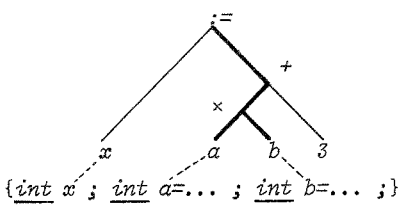
Translating a function is based on the synthetized attributes of the parameters of the function, and on the inherited attributes of the function itself. Moreover, the translation can also take into account all the functions associated to the nodes situated on the path between the node of the current function and the initial node ; this allows us to make previsions on what will happen to the result of that function, and in some cases to generate better code. As we shall see in the next section, a very simple and efficient automaton can be used to implement the above principles.

Example 0.1

Source program :

$$x := a \times b + 3$$

Syntactic tree : {the part of the tree used to translate '×' is bold faced}



Intermediate code :

```
× (proc (int,int)int, access a, access b, access w)
+ (proc (int,int)int, access w, access 3, access w1)
:= (int, access x, access w1)
```

Machine code without local optimizations :

```
LDA access a
MPY access b
STA access w

LDA access w
ADA   = 3
STA access w1

LDA access w1
STA access x
```

Characteristics of the program at different stages of the compilation.

Source language	Result of the syntactic analysis	Intermediate code	Machine code
Semantics - Primitive functions - Primitive data - Composition rules *Syntax* - Means for program representation - Defines a structure reflecting the semantics	The syntactic structure is made explicit : -syntactic tree -links between declared objects and their declaration	Same primitive functions and data as the source language, but -independent modules, the parameters of which are static properties of stored values -interface ensured through (1)storage allocation and (2)sequential elaboration -machine independency	Primitive functions = instructions Primitive data = words, bytes ... -independent modules, the parameters of which are machine addresses -interface ensured through (1)storage allocation and (2)sequential elaboration
-Lexical analysis -Context-free analysis -Declaration handling SYNTACTIC ANALYSIS	-Static elaboration -Storage allocation TRANSLATION PROPER	-Decomposition of source functions and values -Local optimizations	

Machine code with local optimizations :

```
LDA access a
MPY access b
ADA   = 3
STA access x
```

0.2 *THE TRANSLATOR AUTOMATON*

In practice, the syntactic analyzer should deliver a form of tree well suited for the translator automaton ; we propose here a *linear prefixed form* of the tree(†). In this form, the terminals representing declared objects are connected to their declaration by means of a symbol table (*SYMBTAB*). In this table there is one entry for each declaration. For a declared object, both its declaration and applications are connected to the same *SYMBTAB* entry. This allows to make the static properties of the objects, defined at their declaration, available at each of their applications.

The translator automaton scans the linear prefixed form from left to right, accumulating top information on a so called top stack (*TOPST*) and bottom information on a so called bottom stack (*BOST*), while intermediate code is generated. Static properties of declared objects are obtained through *SYMBTAB*. More precisely, the automaton consists of :

(1) *An input tape* containing the source program ; this consists of prefix markers for the nonterminal nodes of the tree, and of basic constructions (i.e. denotations, identifiers ...) for the terminal nodes.

(2) *An output tape* where the intermediate code is generated.

(3) The so called *bottomstack* (*BOST*) where static information is stored in such a way that when an action is translated, the static properties, i.e. the synthetized attributes of its n parameters, can be found in the n top elements of *BOST*.

(4) The so called *topstack* (*TOPST*) containing at each moment the prefix markers and the inherited attributes of the not completely translated actions, in such a way, each time an action is translated, the complete future story of its result can be found on *TOPST*.

(5) The *symbol table* (*SYMBTAB*) where the static properties of each declared object deduced from its declaration are stored in order to be retrieved at each of its application, thus allowing to initialize the process of static elaboration.

(†) In ALGOL 68, coercions are a kind of implicit monadic operators ; in the sequel they will be supposed to have been made explicit by the syntactic analysis [15].

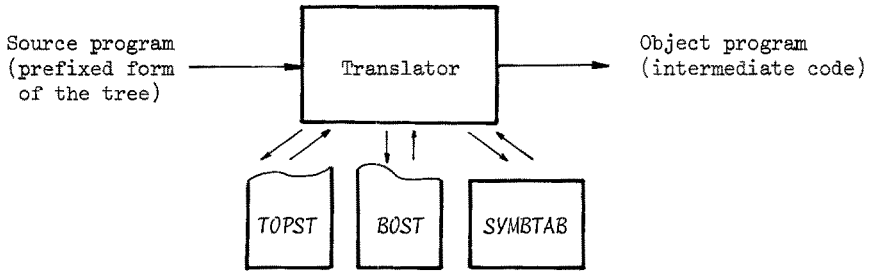


fig. 0.1

The translation of a given action can be separated in three parts :

(1) the *prefix translation* which is performed when the prefix marker of the action is scanned in the source program ; it may consist of the generation of prefix code.

(2) the *infix translation* which is performed in between the translation of two subjacent actions ; it may consist of code generation by which the value of a parameter will be copied at run-time, together with the corresponding updating of the static properties of the parameter at the top of *BOST*.

(3) the *postfix translation* which corresponds to the translation proper of the current action ; it consists of the generation of the corresponding object instructions, together with the replacement, at the top of *BOST*, of the static properties of the parameters of the current action by the static properties of its result (static elaboration).

This is described in a more precise way by the flowchart of fig. 0.2.

PART I is mainly devoted to the description of static properties. Beforehand, the principles of a storage allocation scheme are recalled (I.1).

Example 0.2

Source program :

$$x := a \times b + 3$$

Result of the syntactic analysis :

$$:= \; x \; + \; \times \; a \quad \underset{(1)}{b} \quad \underset{(2)}{3}$$

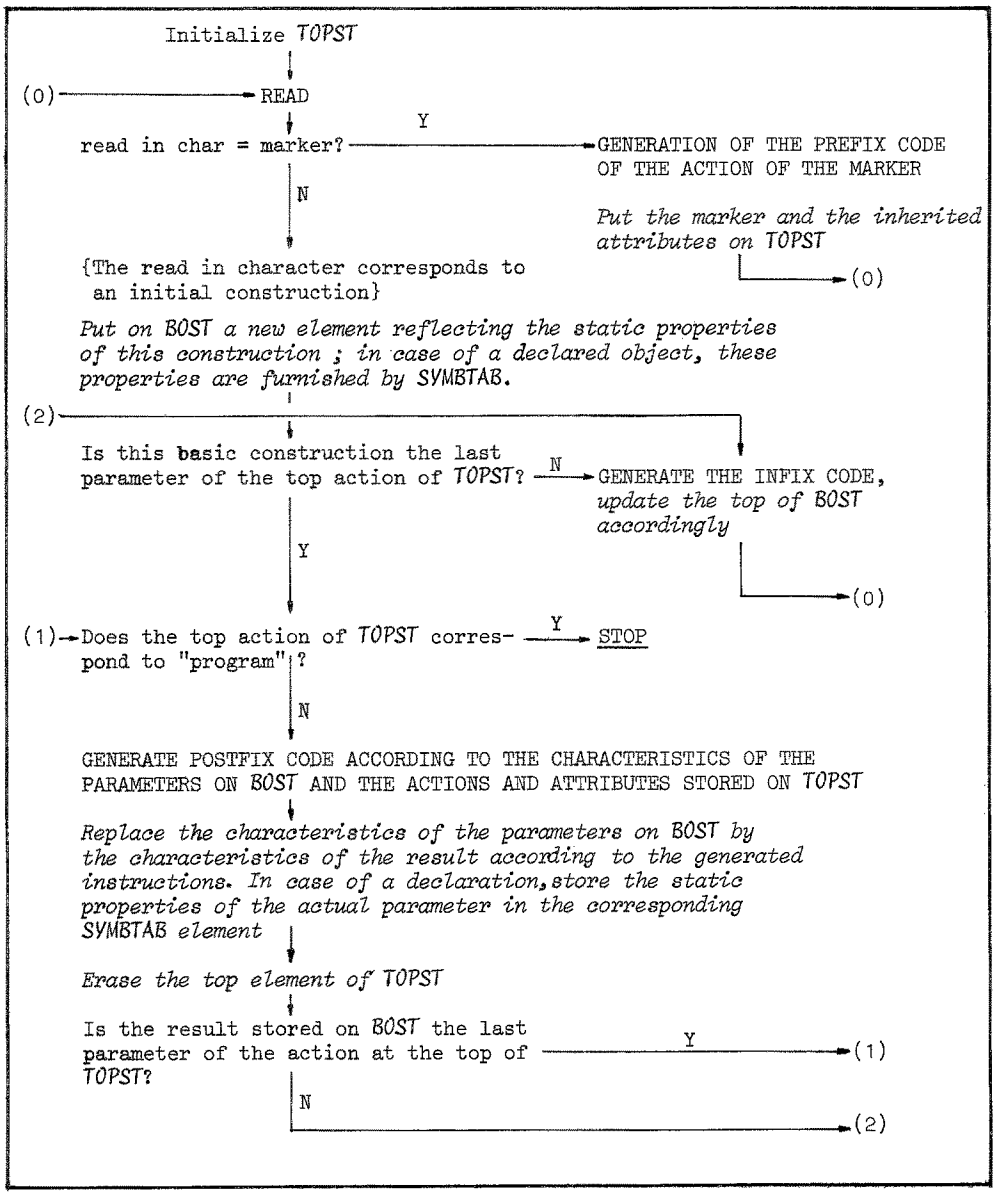
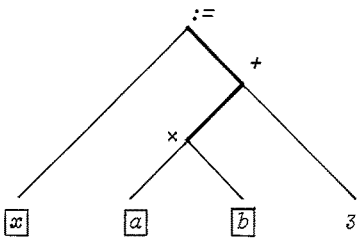


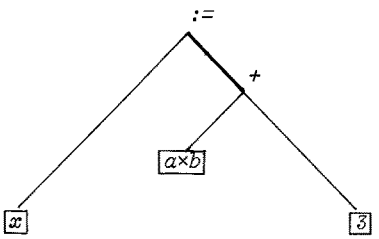
fig. 0.2 : The translator automaton.

(1) Snapshot of the stacks when b is being translated :



BOST □	TOPST
x	:=
a	+
b	×

(2) Snapshot of the stacks when 3 is being translated :



BOST □	TOPST
x	:=
$a \times b$	+
3	