ON COMPILERS BASED ON OBJECT - ORIENTED CONCEPTS

THESIS
SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS
OF THE AWARD
OF THE (M.SC.) DEGREE

519.4
N. M

رسالة

PRESENTED BY
NAGLAA MOHAMED REDA TAHER

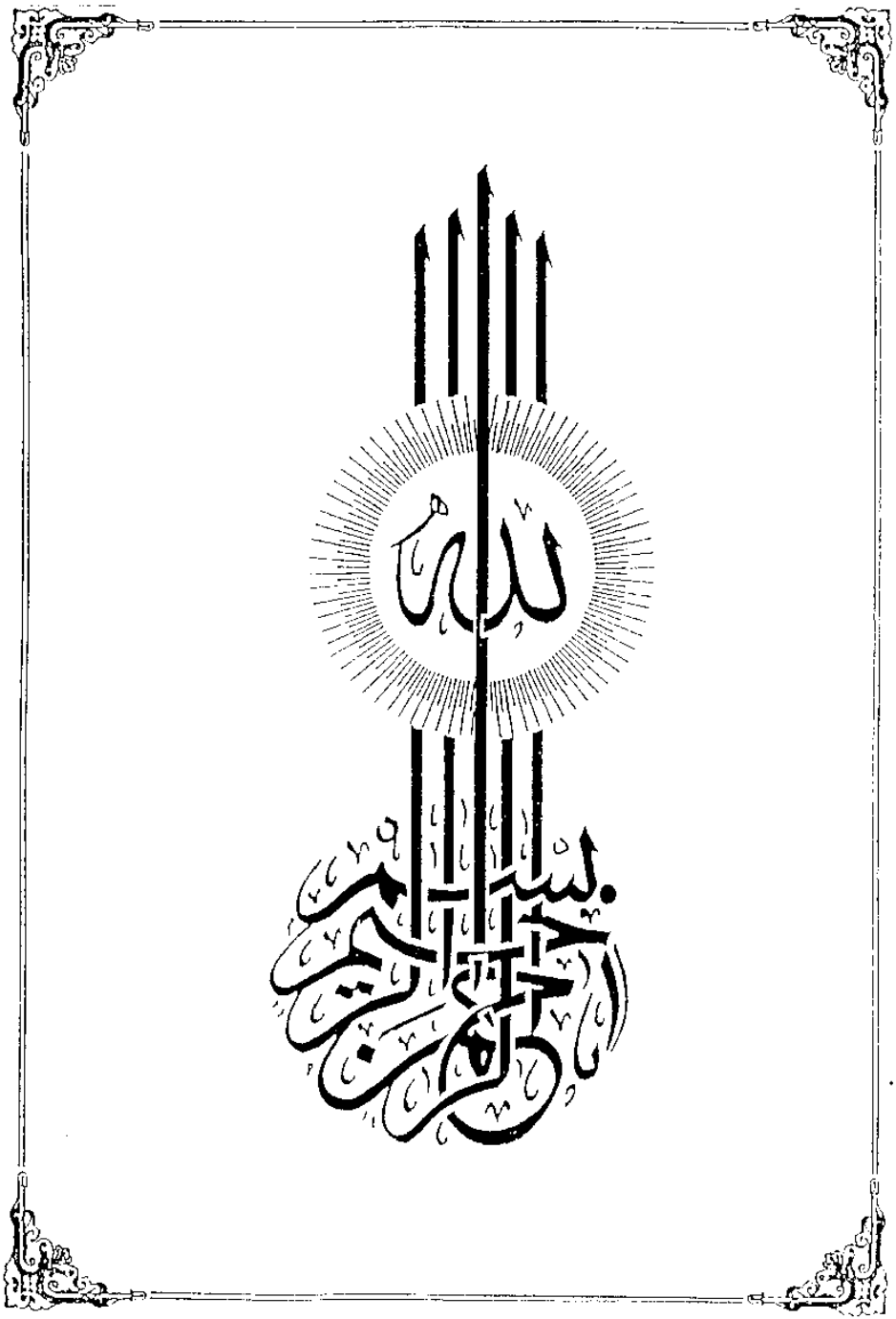
72817

SUPERVISED BY
Dr. Sameh Sami Daoud
Department of Mathematics
Faculty of Scince, Ain Shams University

SUBMITTED TO
Department Of Mathematics
Faculty Of Scince
Ain Shams University
CAIRO, EGYPT
1997

Central Library - Ain Shams University

Central Library - Ain Shams University



Central Library - Ain Shams University

جامعة عين شمس

كلية العلوم

ACKNOWLEDGMENT

*First and foremost, thanks are to **GOD**, the most beneficent and merciful*

I would like to acknowledge my deepest gratitude and thankfulness to associate professor Dr Sameh Sami Daoud, Department of Mathematics, Faculty of Science, Ain Shams University, for suggesting the topic of the thesis, for his patience, for his advice and for giving so generously of his time in reading this work.

Central Library - Ain Shams University

Central Library - Ain Shams University

Central Library - Ain Shams University

the *queues* template class implementing a queue of any type, and explains its usage with Mini-Ada compiler.

Chapter three discusses scanning. It starts by explaining how to translate regular expressions into finite automata. Then it gives another simple and fast technique for scanning. After that, it studies the Mini-Ada scanner which we have designed and shows how it reads and scans the source file. It ends by introducing some scanner generators.

Chapter four studies parsing. It begins by defining context-free grammars, and derivations. Then it discuses the two general approaches of parsing, the top-down and bottom-up parsing and algorithms for each one. Next, it studies the Min-Ada parser which is a sequence of recursive-descent procedures implementing the top-down parsing algorithm with its syntax errors handling. Finally, it discusses the parser generator ANTLR.

Chapter five concerns with the transition of languages with context-free grammars into an intermediate code. It starts by discussing Attribute grammars. Then it introduces two ways for associating semantic rules with productions, transition schemes and syntax-directed definitions. After that, it explains Mini-Ada semantic checking and error messages with two examples, one of an error-free program and the other of a program including semantic errors. It also presents some kinds of intermediate representations. At last, it illustrates a list of all the intermediate code generated {the p-code} by Mini-Ada parser.

Chapter six discusses the synthesis part in our compiler. First, we discuss the basic ideas of the Assembly language which we need. Next, we describe Mini-Ada code generation functions and give a table including all possible intermediate codes and the translated Assembly instruction set.

Central Library - Ain Shams University

CHAPTER ONE
BASIC DEFINITIONS

Central Library - Ain Shams University

Backus-Naur form or BNF- is widely used in language definitions [12] .

The Output.

The output from a compiler comes into two parts -the object program generated and the program listing. Remembering that the compiler deal with more incorrect than correct programs, the qualities of the program listing and error messages produced may be as important as those of the object code [12].

Types Of Compiler.

Compilers may be distinguished according to the kind of machine code they generate. There are three kinds of machine code. Pure machine code that generate code for a particular machine's instruction set. Augmented machine code that generate code for a machine architecture augmented with an operating system routines. Finally, virtual machine code that generate code composed entirely of virtual instructions. Also compilers differs in the format of the target machine code they generate. For example Assembly language format that generates Assembly code, or relocatable binary format that generates a code in a binary format, or memory-image {load-and-go} format that loads the compiled output into the compiler's address space and immediately execute it [4].

Specified Design Constraints.

A compiler may be subject to constraints on its design and implementation. These constraints are reliability, efficiency, and flexibility [12].

Compiler Construction Tools.

There are a number of complex systems, called compiler-compiler, or compiler generators devolved to help construct compilers.

Central Library - Ain Shams University

They are automatic compilers writing systems which produce a complete compiler for any specific language. As an example, see the GAG compilers writing system in [13], or the experimental compiling system ECS in [5].

Some compiler-compiler systems are extended to include the production of optimizers, for example the production-quality compiler-compiler PQCC project, which takes as input the original descriptions of the language and the target machine-dependent information [3].

1.2 THE STRUCTURE OF A COMPILER.

In this section we specify the structure of a compiler by discussing its phases, and how to group them.

The two outputs *errors* and *object code* reflect the compiler's underlying twin purposes which are

(a) To determine if the input program is error free, and to diagnose the errors if there are any.

(b) To generate an equivalent object code program [12].

This leads immediately to a subdivision of the compiler process. Thus any compiler must perform two major tasks: *analysis* of the source program being compiled, and *synthesis* of a machine-language program which, when executed, will correctly perfume the activities described by that source program [4] .

The primary task of the analysis part is to apply the rules of the language definition to the input source program, so determining any errors which exist within it. Our definition distinguishs in form between the {formally expressed} syntax rules and the {informally expressed} semantic rules. Since application of the semantic rules depends on prior application of the syntax rules, a logically and commonly used separation of analyzer activities is as follows:

syntax analyzer (program syntax) (semantic error, analyzed program).

Central Library - Ain Shams University

In practice a further distinguish is usually made within the syntax analyzer between the analysis of those sequences of characters which form the individual symbols of the program, and the analysis of the sequence of symbols itself. The symbol recognition processes usually known as lexical analysis or scanning, so our model of the overall analysis process is now:

Lexical analyzer (character stream) (lexical errors , symbol stream)
syntax analyzer (symbol stream)(syntax errors, program syntax)
semantic analyzer (program syntax)(semantic errors, generated program)
[12].

Lexical Analysis {Scanning}.

The scanner begins the analysis of the source program by reading the stream of characters making up the source program {the input}, character by character, and grouping individual characters into symbols called *tokens*.

The main action of building tokens is often driven by token description. *Regular expression* notation is an active approach to describing tokens [4].

Syntax Analysis {Parsing}.

Given a formal syntax specification, the parser reads and groups tokens hierarchically into grammatical phrases. During the parsing process, the parser verifies correct syntax and if a syntax error is found, it issues a suitable diagnostic. The hierarchical structure of a program is usually expressed by recursive rules. *Context-free grammars CFG* are a formalization of recursive rules that can be used to guide syntactic analysis [4].

Semantic Analysis {Semantic Routines}.

Semantic routines perform two functions: First, they check the static semantics of each construct. Second, they generate an explicit

Central Library - Ain Shams University

intermediate representation IR of the source program which implements the construct [4].

The Code Optimization.

Some compilers attempts to improve the intermediate code, so that faster running machine code will result. Since the code produced by straightforward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations [1].

The Code Generation.

The main task of the synthesis part is the generation of target code, consisting normally of relocatable machine code or Assembly code, this task is called code generation [1].

The Grouping Of Phases.

As a result of the above discussion, a compiler operates in phases, each of which transforms the source program from one representation to another. A way to simplify compiler structure, if no optimization is required, is to merge code generation with semantic routines and eliminate the use of an IR. The result is a *one-pass* compiler. So, in implementation, activities from more than one phase are often grouped together.

Often, the phases are collected into a *front end* and a *back end*. The front end consists of those phases, or part of phases, that depend primary on the source language and are largely independent of the target machine.

The back end includes those portions of the compiler that depend on the target language, and generally, these portions do not depend on the source language, just the intermediate language [1].

Our Mini-Ada compiler is essentially a parser with subordinate front end and back end functions. The front end reads and scans

Central Library - Ain Shams University

source code, written in Mini-Ada language, for the parser while the back end expresses the out code in Assembly language.

1.3 MINI-ADA LANGUAGE.

In this section we introduce a complete definition of our source language, the Mini-Ada language, in a natural order, and gives all information we need about it.

Mini-Ada is a subset of Ada. We shall provide the definition of the language we use as a source language for our compiler implementation project.

The following is an extended BNF grammar for Mini-Ada. Nonterminal symbols are enclosed in < and >. Reserved words are in bold, token classes are undelimited words, and all other symbols represent themselves. The symbol → separates the left- and right-hand side of a production. if no left-hand side is shown, the left-hand side of the preceding production is assumed.

< Compilation >	→	<Package-Dec>
< Package-Dec >	→	**Package** < Package-Spec >
< Package-Spec >	→	<Id> **is** {< Dec >} < Body-Opt > **end;**
< Body-Opt >	→	[**Body** {< Body-Dec >} {<Statement >}]
< Body-Dec >	→	< Subprogram-Body-Dec >
	→	< Dec >
< Dec >	→	<Obj-Dec >
	→	< Type -Dec>
	→	< Supprogram-Dec >
<Object-Dec>	→	< Id-List >: <Type >
< Id-list >	→	< Id > {, < Id>}
< Id >	→	Identifier
< Type-Dec >	→	<Type><Id>is<Type-Def>;
< Type >	→	A-type
	→	<Array-Type-Def>

Central Library - Ain Shams University

< Array-Type-Def >	→	Array <Constrained-Index-List> of <Type>
< Cnstrained-Index-List >	→	(<Range>{,<Range>})
<Range >	→	int-literal .. int-literal
<Subprogram-Dec>	→	<Subprogram-Spec> ;
<Subprogram-Body-Dec>	→	<Subprogram-Spec> is <Body-Dec>} Begin {<Statement>} end;
< Subprogram-Spec >	→	Function <Id > <Formal-Part-Opt>
	→	procedure <Id> <Formal-Part-Opt>
<Formal-Part-Opt >	→	[(< Parameter-Dec-List >)]
< Parameter-Dec-List >	→	<Parameter-Dec>{;<Parameter-Dec>}
< Parameter-Dec>	→	<Id-List>:<Type>
<Statement>	→	< Assignemt-St >
	→	< Call-St >
	→	<Loop-St >
	→	<If-St >
	→	< Retun-St >
	→	< Case-St >
	→	< Exit-St >
<Assignment-St >	→	< Id > := < Simple-exp >;
<Call-St >	→	< Id > ;
<Return-St >	→	Return [< Simple-Exp >] ;
<If-St >	→	If < Bexp > Then { <Statment > } [Elsif < Bexp >Then { <Statment>}] [< Else-Part >] End;
<Else-Part >	→	Else { < Statment > }
<Loop-St >	→	[< Iteration >] <Basic-Loop >;
<Basic-Loop >	→	Loop { < Statment > } End loop
< Iteration >	→	While < Bexp >
	→	For <Id > In <Range >
<Exit-st>	→	Exit [When <Bexp>] ;

Central Library - Ain Shams University

<Case-ST >	→	Case < Simple-exp >Is < When-List > < Others-Opt >End case ;
< When-List >	→	{<Simple-exp> =>{<Statement >} }
<Others-Opt >	→	[When others =>{ < Statement >}]
< Bexp >	→	< Rel > { < Logical-Op > <Rel > }
< Rel >	→	[<Simple-exp> Relational-op <Simple-exp>]
< Simle-Exp >	→	<Term >{Adding-op<Term >}
<Term >	→	<Factor>{Multiplying-op<Factor >}
< Factor >	→	<Primary >
	→	not<Primary >
<Primary >	→	< Literal>
	→	Identifier
< Literal>	→	Int-literal
	→	Float-literal
	→	Character
<Logical-Op >	→	and
		or

Tokens.

Since tokens are the building blocks of programs, we begin our study of the Mini-Ada language by defining its tokens.

keywords {reserved words} are the tokens that look like words or abbreviations and serve to distinguage between the different language constructs, they are:
and, array, begin, body, case, else, elseif, end, exit, for, function, if, in, is, loop, not, of, or, others, package, procedure, return, then, type, when, while.

Literals are integer, character, float. Integer literals contain only digits and underscores and start with a digit. Real literals contain a decimal point, with at least one digit both precede and follow it. Character literal begins and ends with a single quote(') and contain only one printable character.

Central Library - Ain Shams University