

OPERATOR OVERLOADING IN C

by

William S. Miles

BSc(CS) — University of British Columbia

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Master of Science
in the
Faculty of Computer Science

This thesis is accepted.

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

March, 1995

© William S. Miles, 2004

Abstract

This thesis is concerned with investigating an approach for adding problem notation to a computer language. A method is proposed for modifying a programming language to fit problem notation by including the capability of defining a problem specific set of symbols or operators in the language.

The proposed method for extending a language uses operator overloading to map language operations to the problem notation. This method is shown to be feasible for the case where the language is defined through a context free grammar and where the problem abstractions are implemented through object services or functions. We also show how generalized operator overloading can be used to resolve some of the language difficulties implicit in C, which cannot be done through C++ operator overloading.

A preprocessor for C was built to study the problems and effects of adding generalized operator overloading to a language. Experience with this method suggests that problems may occur when encoding the semantic variations for a problem service, selecting unambiguous operator symbols, and implementing general language extensions for assignment operations and identifier references. Host language support for reference variables is necessary.

Our results indicate that generalized operator overloading can be used to provide a domain specific problem notation that is user defined, but processed by C code.

Contents

Abstract	ii
List of Tables	vi
List of Figures	vii
Acknowledgements	viii
1 Introduction	1
1.1 Programming Languages	2
1.2 Language Notation	2
1.3 Language Adaptation	4
2 Language Fundamentals	6
2.1 Language Theory	7
2.2 Semantic Functions	9
2.3 Operator Symbols	13
2.4 Operator Precedence	14
2.5 Summary	17
3 Expression Evaluation	19
3.1 Grammar Specification	20
3.2 Sentence Evaluation	21

4	Operator Overloading	26
4.1	Operator Overloading in C++	27
4.2	Operator Classification	29
4.2.1	Assignment and Initialization	30
4.2.2	Indexing and Name Qualification	31
4.3	Function Overloading	31
4.4	Semantics of Overloaded Operators	32
4.5	Operator Modification	33
4.6	Summary	34
5	Implementation Design	36
5.1	Conceptual Model	37
5.2	Grammar Specification	38
5.3	Grammar Extensions	41
5.4	Language Preprocessor	44
5.4.1	Grammar Input Module	44
5.4.2	Lexical Analysis Module	47
5.4.3	Language Parse Module	48
5.5	Operator Interface Library	48
6	Results	52
6.1	APL Functions in C	53
6.2	Logic Operators in C	56
6.3	An Improved C	59
6.4	The Poetry of C	61
7	Conclusions	64
	Bibliography	68

Appendices

I	Grammar for C	70
II	Font Table	78

List of Tables

3.1	Evaluation Levels	25
5.1	Operator Semantic Flags	42
6.1	APL Operator Symbols	53
6.2	APL Operator Extensions for C	54
6.3	C Operator Redefinition	59
6.4	C Operator Semantic Interpretation Definition	60

List of Figures

2.1	Operator Precedence Example	17
2.2	Operator Precedence Derivation Tree	18
3.1	A Generalized Expression Grammar	20
3.2	An Abbreviated Expression Grammar Example	22
3.3	Algorithm 1 - Expression Evaluation	23
4.1	A Generalized Grammar for Operator Modifiers	34
5.1	Conceptual Model for Generalized Operator Overloading	38
5.2	A Grammar Syntax Specification	40
5.3	Language Preprocessor Architecture	44
5.4	Algorithm 2 - Grammar Construction	46
5.5	Algorithm 3 - Lexical Analysis	47
5.6	Algorithm 4 - Recursive Parsing	49
5.7	Parse Path Example	50
5.8	Class Interface Library Example	51
6.1	A Grammar for APL Expressions in C	54
6.2	APL Operators in C	55
6.3	Logic Operator Grammar Rules	56
6.4	Logic Operators in C	58
6.5	New C Operators	60
6.6	A Poetic C Program	62

Acknowledgements

I am very grateful to the following people for their assistance and encouragement: Professor LeRoy Johnson, for his direction and encouragement of this work; Dr. Joseph Horton, who refreshed my interest in mathematics; Skip Ciampa, for his support during my employment; and to my family, who provide the motivation for striving for excellence: Marian Miles, William Miles, Shaun Miles, and Marie Miles.

Chapter 1

Introduction

“I would have wished to present it to you naked and unadorned, without the ornament of a prologue or the countless train of customary sonnets, epigrams, and eulogies it is the fashion to place at the beginning of books. For I can tell you that much labour though it cost me to compose, I found none greater than the making of this preface you are reading. Many times I took up my pen to write it, and many times I put it down, not knowing what to say.”

Don Quixotes, Cervantes

1.1 Programming Languages

A major difficulty encountered by every user of a programming language is that the familiar is replaced by the unfamiliar. Programmers who already know the problem specification in their domain of understanding spend much time translating this problem specification into a programming language specification [14, 20]. Yet most programming languages are already much too complex, in their full implication, for many programmers. Thus, they are more than unsuitable for casual use. Programming languages are often designed to satisfy many poorly defined and often conflicting objectives, such as efficiency, generality, problem-orientedness, machine independence, compatibility, reliability, ease of programming, and so on. Not surprisingly, a language's ability to deal with problem oriented specifications is often less than satisfactory.

Every programming language provides some capability for capturing problem domain semantics explicitly. A language that captures more of the problem domain semantics is very significant, for the reasons of providing an underlying uniform representation to the problem solution, reusability of functions, and improved maintainability of application code [4, 18]. It is therefore remarkable that so little has been written from a language independent point of view about user modification of the language design; yet it seems reasonable to expect that investigation into how programming languages might be systematically adapted will lead to better languages.

1.2 Language Notation

Language notation pertains to the formal specification of a language grammar and the semantics of the terms used in the language vocabulary. Much work by Knuth, Chomsky, Aho, and others [9, 10, 23] assisted in developing a formal basis for understanding language theory. However, our experience has shown that many languages

evolve over time to meet new needs. Various programming language extensions using such techniques as preprocessors and macro facilities are often provided to programmers dissatisfied with their standard language facilities. In many cases, the intent of a language extension is to define a notation which allows convenient expression not only of concepts arising directly from a problem, but also of those arising in subsequent problem analysis, generalization, and specialization.

For problem solving, mathematical notation provides perhaps the best known and best developed example of language used to consciously express thought in a clear and concise fashion [3, 11, 12, 17]. Unfortunately, mathematical notation suffers from serious deficiencies through lack of universality and the need to interpret mathematics differently according to the topic and immediate context. Programming languages, however, represent precise and unambiguous language implementations because they are designed specifically to direct the operation of a computer. They are often intended to be universal (general-purpose), yet this universality constrains the language in ways that make most programming languages decidedly inferior to mathematical notation for concise expression of thought.

Two major weaknesses of language design are a strong tendency to repeat semantically previous languages while making minor modifications in syntax, followed by a tendency to limit syntax to the standards defined for similar languages. There are good historical reasons for this, but the state of the art no longer needs such restriction.

This thesis is based on the idea that the central functions of a programming language are: to provide a language facility for the logical specification and execution of code; to provide constructs for the declaration and maintenance of program entities or variables; and to perform expression evaluation. In the latter case, language operators or functions are characterized by the problem and need not be restricted to a predefined set of arithmetic, logical, and referential symbols declared within the language. This thesis proposes to investigate a method of capturing problem domain

semantics explicitly by defining a problem syntax representation directly within the programming language.

1.3 Language Adaptation

The utility of adapting a programming language, as we propose, is based on the idea that an object is an entity which has meaning within the problem domain [5, 15, 21, 25]. Object oriented design is a discipline which focuses on the encapsulation of data structure and methods in a single conceptual entity, the object. An object, as an instance of its class declaration, is described in terms of its attributes and services without consideration for the specific details of its implementation. When an object is used as a program entity, it becomes an abstraction of some part of the problem domain. The operations appropriate for the object can now be considered as a concise notation or algebra specific to the problem. These operations should be directly expressible within the programming language.

An object has services or functions which it can perform. Operator overloading, considered as a method for mapping object services to language operators, is examined as a solution to the problem of extending a programming language to include a problem specific notation. Operator overloading is now in common use but often in a quite restricted way. For instance, the C++ language cannot be extended by creating new operators, operators cannot be redefined from unary to binary types, operator precedence cannot be altered, operator associativity cannot be changed, and operator meaning is defined by the language for built-in data types. These restrictions limit the implementation of problem notation in the programming language.

This thesis is organized as follows. Chapter 2 is a review of language theory, and provides the background of a language grammar and an algebra in which the problem notation is defined. The semantics of an operator symbol are defined in terms of an interpretation function, and an operator precedence hierarchy is understood in

terms of a derivation tree. Chapter 3 describes our grammar for formulating expressions in a language and presents our algorithm for evaluating expressions in terms of the algebra semantic interpretation functions. Chapter 4 provides an overview of operator overloading, with specific references to the limitations that currently exist in the C language. Generalized operator overloading is discussed as an approach to extending a language grammar. This chapter classifies C operators in preparation for mapping the algebra of these operators to the restricted language grammar introduced in Chapter 3. Chapter 5 discusses the implementation design for our model of generalized operator overloading. This model is implemented for the C language. A new extension to C is developed, which provides a solution to the problem of introducing new operator symbols into the language. We describe the development of our language preprocessor that accepts a language grammar containing new operator symbols and translates the operator symbols into function calls. Chapter 6 illustrates some results of our solution. As an example, vector arithmetic is introduced to the C language through the use of APL operators in C. Another example redefines the C language operator symbols to alleviate some of the more common programming errors inherent in the language. We present a curious application of our method by introducing the idea of compiling poetic programs. We conclude in Chapter 7 with an appraisal of our approach and design methodology.

Chapter 2

Language Fundamentals

“To sum up, pictographic and hieroglyphic writing as used in Babylonian, Mayan, and Chinese cultures represents an extension of the visual sense for storing and expediting access to human experience. All of these forms give pictorial expression to oral meanings. As such, they approximate the animated cartoon and are extremely unwieldy, requiring many signs for the infinity of data and operations of social action. In contrast, the phonetic alphabet, by a few letters only, was able to encompass all languages. Such an achievement, however, involved the separation of both signs and sounds from their semantic and dramatic meanings. No other system of writing has accomplished this feat.”

Understanding Media, Marshall McLuhan

2.1 Language Theory

The modern definition of a programming language includes a specification of the language syntax and its semantics [8, 9, 10, 16]. A language *grammar* defines a set of production rules, which specify how language statements and expressions are formed from the fundamental terms in the language vocabulary.

A finite set of symbols is called an *alphabet* or *vocabulary*. The terms *symbol*, *token* and *word* are considered synonymous and denote an element of a vocabulary. If V is a vocabulary, a finite sequence of symbols in V is called a *string* over V . The notation V^* denotes the (infinite) set of all strings over V , including the empty string, denoted ε . The notation V^+ denotes $V^* - \varepsilon$. If x is a string, then $|x|$ denotes the *length* or number of symbols in x .

Definition 1 A *grammar* is a quadruple $G = \{N, T, P, \Phi\}$ where:

- N is a finite set of symbols called *nonterminals*.
- T is a finite set of symbols called *terminals* such that N and T are disjoint, and V denotes the union of N and T ($V = N \cup T$).
- Φ is a distinguished element in N called the *start symbol* or *goal*.
- P is a finite set of pairs called *productions*. Each production (α, β) is written $\alpha \rightarrow \beta$. The left part α is in V^+ and contains at least one nonterminal. The right part β is in V^* .

A grammar $G = \{N, T, P, \Phi\}$ is a *context free grammar* (abbreviated CFG) if each production rule is of the form $A \rightarrow \alpha$, where $A \in N, \alpha \in V^*$. The term “context free” means that A can be replaced by α wherever it appears, no matter the context.

A Backus-Naur (BNF) specification is a notation used to represent grammar production rules [16]. The form uses four meta characters which are not in the working vocabulary. These are: \langle , \rangle , $::=$, and $|$. The idea is that strings, which do not contain the meta characters, are enclosed by \langle and \rangle and denote elements of N . The symbol $::=$ serves as a replacement operator like \rightarrow , and $|$ is read “or”.

For example, an ordinary context free grammar for unsigned digits in a programming language might be as follows, where D stands for the class of digits, and U stands for the class of unsigned integers:

$$\begin{array}{lll}
 D \rightarrow 0 & D \rightarrow 4 & D \rightarrow 8 \\
 D \rightarrow 1 & D \rightarrow 5 & D \rightarrow 9 \\
 D \rightarrow 2 & D \rightarrow 6 & U \rightarrow D \\
 D \rightarrow 3 & D \rightarrow 7 & U \rightarrow UD
 \end{array}$$

This example, when written in BNF, becomes:

$$\begin{array}{l}
 \langle \text{digit} \rangle ::= 0|1|2|3|4|5|6|7|8|9 \\
 \langle \text{unsigned-integer} \rangle ::= \langle \text{digit} \rangle | \langle \text{unsigned-integer} \rangle \langle \text{digit} \rangle
 \end{array}$$

In this thesis, we will adopt a modified form of the BNF notation. We shall use \langle and \rangle to denote nonterminal grammar symbols, but not the other meta characters. The modified form describes, explicitly, the set of production rules used for a nonterminal symbol. This set, along with the symbol \rightarrow , retains the basic idea that a nonterminal symbol can be replaced by any member of its production rule set.

Production rules will contain a nonterminal grammar symbol on the left hand side and a list of terminal or nonterminal symbols on the right hand side. Where a

nonterminal symbol on the right hand side is optional, in that it may or may not be present in the production, this is denoted with the subscript *opt* being attached to the symbol. The use of optional symbols can simplify the production rule notation. Where more than one syntactic production rule is possible for a left hand side symbol, each rule is independently specified.

This thesis introduces a symbolism and a notation for specifying expressions within the programming language. The language grammar specifies how expressions are derived and how operator symbols can be used within the language. If $\alpha \rightarrow \beta$ is a production and $\gamma\alpha\delta$ is a string then $\gamma\beta\delta \Rightarrow \gamma\alpha\delta$ is an *immediate derivation*. A *derivation* is a sequence of strings $\alpha_0, \alpha_1, \dots, \alpha_n$ such that $\alpha_0 \Rightarrow \alpha_1, \alpha_1 \Rightarrow \alpha_2, \dots, \alpha_{n-1} \Rightarrow \alpha_n$. Any string α derivable from the start symbol Φ of G containing only terminal symbols is called a *sentence* generated by G .

For example, consider the following CFG which describes a simplified version of arithmetic expressions:

$$G = (\{E, T\}, \{+, (,), a\}, P, E)$$

where the production rules P are:

$$\begin{aligned} \langle E \rangle &\rightarrow \langle E \rangle + \langle T \rangle \\ \langle E \rangle &\rightarrow \langle T \rangle \\ \langle T \rangle &\rightarrow (\langle E \rangle) \\ \langle T \rangle &\rightarrow a \end{aligned}$$

The expression $a + (a)$ is a sentence derivable from G according to the sequence $E \Rightarrow E + T \Rightarrow E + (E) \Rightarrow E + (T) \Rightarrow E + (a) \Rightarrow T + (a) \Rightarrow a + (a)$.

2.2 Semantic Functions

The semantics of an operator define the operator meaning in context with the environment in which the operator is used. Given a set $\Sigma \subseteq T$ of operator symbols, it

is desirable to have a standard set of symbols to name operations taking a specified number of arguments. Such a set is called a *ranked alphabet*.

Definition 2 Let \mathcal{N} be the set of natural numbers. A *ranked alphabet* is a set Σ with an associated function $r : \Sigma \rightarrow \mathcal{N}$ assigning a *rank* or *arity* n to each symbol f in Σ . For every $n \geq 0$, Σ_n denotes the subset of Σ consisting of the operator symbols of rank n . For example, the set of constants is Σ_0 . The set of unary operator symbols is Σ_1 .

The environment in which an operator has meaning is called an *algebra*. An algebra is simply a pair (A, F) consisting of a nonempty set A together with a collection F of functions, also called operations, each function in F being of the form $f : A^n \rightarrow A$, for some natural number $n > 0$. Given an algebra (A, F) , each symbol f in Σ receives an *interpretation* $I(f)$ which is a function in F . In other words, the set F of functions is defined by an interpretation $I : \Sigma \rightarrow F$ assigning a function of rank n to every symbol of rank n in Σ . More formally,

Definition 3 Given a ranked alphabet Σ , a Σ -*algebra* \mathcal{A} is a pair (A, I) where A is a nonempty set called the *carrier*, and I is an *interpretation function* assigning functions to the symbols as follows:

- Each symbol f in Σ of rank $n > 0$ is interpreted as a function $I(f) : A^n \rightarrow A$
- Each constant c in Σ is interpreted as an element $I(c)$ in A .

Informally, the ranked alphabet describes the syntax, and the interpretation function I describes the semantics.

For example, let A be the set $\{0, 1\}^*$, and $\Sigma = \{0, 1, +\}$, where $0, 1$ are constants of rank 0, and $+$ has rank 2. If we define the interpretation function I such that $I(0) = 0, I(1) = 1, I(+)$ = concatenation, we have an algebra of binary numbers

$\mathcal{A} = (\mathcal{A}, \mathcal{I})$. The semantic interpretation of the $+$ symbol is understood to be the concatenation of two binary numbers into a new binary number.

To review, operator semantics are determined by the interpretation function associated with the operation. This function is valid only within certain contexts or domains. The context is determined by the domain and range of the semantic function, as specified by the data type of the carrier set used for function arguments and the function result.

When dealing with multiple domains, it is useful to generalize algebras by allowing domains and operations of different types. These are called *sorts*. Let S be the set of sorts, or types. Typically, S consists of types in a programming language (such as *integer, real, boolean, character, etc.*). The notion of an algebra is extended into a *many-sorted algebra* as follows:

Definition 4 An *S-ranked alphabet* is a pair (Σ, r) consisting of a set Σ together with a function $r : \Sigma \rightarrow S^* \times S$ assigning a rank (u, s) to each symbol f in Σ . The cardinality of the string u in S^* is the *arity* of f and s is the *sort* or *type* of f . If $u = s_1 \cdots s_n, (n \geq 1)$, a symbol f of rank (u, s) is to be interpreted as an operation taking n arguments, the i -th argument being of type s_i , with the result being of type s .

Definition 5 Given an S-ranked alphabet S , a *many-sorted Σ -algebra* \mathcal{A} is a pair (A, I) , where $A = (A_s)_{s \in S}$ is an S -indexed family of nonempty sets, each A_s being called a *carrier of sort s* , and I is an *interpretation function* assigning functions to the function symbols as follows:

- Each symbol f of rank (u, s) where $u = s_1 \cdots s_n$ is interpreted as a function $I(f) : A_{s_1} \times \cdots \times A_{s_n} \rightarrow A_s$
- Each constant c of sort s is interpreted as an element $I(c)$ in A_s

To clarify some of the concepts introduced, an example of the algebra of stacks of real numbers can be defined as follows. First, some notation. Let $N_n = \{1, 2, \dots, n\}$ denote the set of natural numbers from 1 to n , and let \mathcal{R} be the set of real numbers. Let $S = \{\mathbf{real}, \mathbf{stack}\}$, $\Sigma = \{\odot, \star, TOP, PUSH, POP\}$. \odot is a constant of type **stack** and \star is a constant of type **real**. TOP is a function symbol of rank $(\mathbf{stack}, \mathbf{real})$, POP is a symbol of rank $(\mathbf{stack}, \mathbf{stack})$, and $PUSH$ has rank $(\mathbf{real}, \mathbf{stack}, \mathbf{stack})$. A stack is defined by any function of the form $S : N_n \rightarrow \mathcal{R} \cup \{\star\}$, the set of all these functions is the carrier of the sort or type **stack**. Similarly, the carrier of the sort or type **real** is $\mathcal{R} \cup \{\star\}$. The function $TOP(S)$ takes a stack S as an argument and returns a real number $S(n)$ from the top of S if $n > 0$, or the symbol \star if $n = 0$. If $\alpha \in \mathcal{R}$, $PUSH(\alpha, S)$ returns the stack $S' : N_{n+1} \rightarrow \mathcal{R} \cup \{\star\}$ such that $S'(k) = S(k)$ for all $k, 1 \leq k \leq n$, and $S'(n+1) = \alpha$. The function $POP(S)$ returns the stack $S' : N_{n-1} \rightarrow \mathcal{R} \cup \{\star\}$ such that $S'(k) = S(k)$ for all $k, 1 \leq k \leq n-1$, if $n > 1$; otherwise if $n = 1$, $POP(S)$ returns the empty stack \odot .

The semantics of an operator symbol in a programming language have been introduced in terms of an interpretation function defined over a set of carriers of various sorts. When the operator symbol is used in a larger context, such as an expression generated through a language grammar, there is now the problem of ensuring semantic consistency between symbols.

A *compound expression* is defined as an expression containing more than one symbol from Σ . Such an expression might be generated through a language grammar given by the two production rules $S \rightarrow \alpha S$ and $S \rightarrow \gamma$, where $\alpha, \gamma \in \Sigma$. Semantic consistency between function symbols can require that intermediary expression results be converted from one sort to another. Automatic type conversion or *coercion* often exists in many programming languages, and it is used as a programming aid to convert between language data types. Coercion establishes a form of semantic consistency for the language operators, within the set of predefined data types.

This thesis does not address type conversion or coercion. Operators or functions can clearly be introduced to the algebra to provide type conversion where necessary. Conversion between sorts or defined data types must therefore be specified for all cases where operators are referenced in compound expressions.

2.3 Operator Symbols

The language vocabulary is the set of fundamental symbols from which sentences are created. The set T defined the language vocabulary. Elements in the vocabulary are known as *tokens*. A typical programming language may have a terminal vocabulary set $\{\text{begin, end, (,), *, +, -, while, \dots}\}$. Each token is constructed as a string derived from some alphabet. A symbol in the vocabulary is defined as a *regular expression* over the alphabet.

Definition 6 Let Σ be a finite alphabet. *Regular expressions* over Σ are defined inductively as follows:

1. \emptyset is a regular expression representing the empty set of strings.
2. ϵ is a regular expression representing the set consisting of the empty string ϵ .
3. for each $\alpha \in \Sigma$, α is a regular expression representing the set $\{\alpha\}$
4. if α and β are regular expressions representing sets A and B , then
 - (a) $\alpha + \beta$ is a regular expression representing $A \cup B$.
 - (b) $\alpha\beta$ is a regular expression representing the language $AB = \{xy : x \in A, y \in B\}$.
 - (c) α^* is a regular expression representing $A^* = \bigcup_{i=0}^{\infty} A^i$, where $A^0 = \{\epsilon\}$ and $A^{i+1} = A^i A$ for $i \geq 0$.

A regular expression is any finite sequence of symbols taken from the alphabet. For example, if the character $+$ exists in the alphabet, then the strings $+$ and $++$ are both regular expressions. Let $+$ be the function symbol whose semantic interpretation

is to define a positive number, and let $++$ be the function symbol whose semantic interpretation is to increment a number. Both function symbols, by definition, are members of the language vocabulary set V . Given any string s such that $++$ is in s , it is clearly ambiguous as to whether the string contains two occurrences of the $+$ function, or one occurrence of the $++$ function.

As this thesis is investigating the introduction of new operator symbols into a programming language, these new symbols can potentially conflict with the existing language vocabulary and result in an ambiguous grammar. We consider this to be an error in the algebraic specification of the grammar symbols. However, a practical solution to this problem is to minimize any ambiguity during sentence derivation by assuming that the lexical scan returns the longer symbol string as the operator token, or by accepting the first derivation of the term as the correct grammatical interpretation of the symbol.

2.4 Operator Precedence

The precedence of language operators is the operator evaluation order as defined by the algebra of the problem and formally implemented through the language grammar. The derivation of a sentence or expression in the language identifies the expression components and evaluation order according to the grammar derivation sequence. The definition of a *tree* and an associated *derivation tree* are as follows:

Definition 7 Let N_+ denote the set of positive natural numbers, and N_+^* the set of all strings of positive natural numbers. A *tree domain* D is a nonempty subset of N_+^* satisfying the conditions:

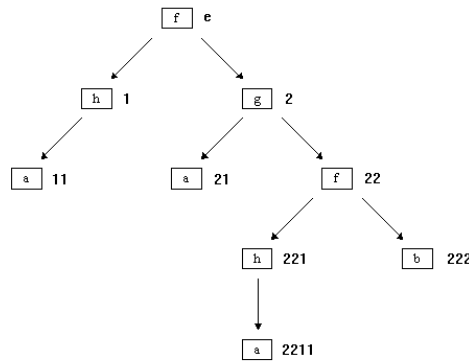
1. For each $u \in D$, every prefix of u is also in D .
2. For each $u \in D$, for every $i \in N_+$, if $ui \in D$ then, for every j , $1 \leq j \leq i$, uj is also in D .

Definition 8 Given a set Σ of labels, a Σ -tree (or tree) is a function $t : D \rightarrow \Sigma$, where D is a tree domain denoted by $dom(t)$, and such that for every node u in $dom(t)$, $d(u) = r(t(u))$, where $d(u)$ is the cardinality of the set $\{i | ui \in dom(t)\}$. Every string u in $dom(t)$ is called a *tree address* or *node*.

The preceding definitions introduced the concept of a tree domain as a set of addresses. A tree is constructed by mapping addresses from D to symbols, such that the rank of a symbol equals the number of immediately subordinate addresses. As an example [9], let $\Sigma = \{f, g, h, a, b\}$. A tree is defined by the function $t : D \rightarrow \Sigma$, where $D = \{e, 1, 2, 11, 21, 22, 221, 222, 2211\}$, and t is the function defined by the pairs

$$\{(e, f), (1, h), (2, g), (11, a), (21, a), (22, f), (221, h), (222, b), (2211, a)\}$$

The graph of this tree is shown below:



A grammar production can now be associated with a tree. It is assumed, without further definition, that the terms: root, node, leaf, subtree, path, and a left to right ordering of the leaves of a tree are understood. Let $u_{\mathcal{T}}$ represent the address of the root node of a subtree \mathcal{T} with m leaves.

Definition 9 Let $G = \{N, T, P, \Phi\}$ be a context free grammar and let \mathcal{T} be a Σ -tree with labelling $t : \text{dom}(\mathcal{T}) \rightarrow N \cup T$. \mathcal{T} is said to be a *grammatical tree of G* if and only if, for every subtree \mathcal{T}' of \mathcal{T} , there exists a production $p \in P$ of the form $\alpha \rightarrow \beta$ corresponding to \mathcal{T}' such that $t(u_{\mathcal{T}'}) = \alpha$ and, for every $b_i \in \beta$, there exists a node $ui \in \text{dom}(\mathcal{T}')$ such that $t(ui) = b_i$.

Definition 10 A *derivation tree* is a grammatical tree \mathcal{T} such that for every tree address u of $\text{dom}(\mathcal{T})$ where $d(u) = 0$ (a *leaf*), ordered from left to right, the string defined by the concatenation of the labels of the leaves $fr(\mathcal{T}) = t(u_1)t(u_2) \cdots t(u_m)$ is in \mathcal{T}^* . The string $x \in \mathcal{T}^*$ is said to be in the language $L(G)$ defined by G if and only if a derivation tree \mathcal{T} exists such that $fr(\mathcal{T}) = x$. The grammar G is *unambiguous* if, for any two derivation trees \mathcal{T} and \mathcal{T}' , $fr(\mathcal{T}) = fr(\mathcal{T}')$ implies $\mathcal{T} = \mathcal{T}'$.

A derivation tree has symbols from the terminal vocabulary set at the leaves of the tree ordered such that the sentence generated from the grammar can be constructed by traversing the tree from left to right, concatenating all symbols at the leaves together.

The following grammar, shown in Figure 2.1, implicitly defines a higher precedence for multiplication over addition.

The parse of the expression $a * b + c * d + e * f * g$ isolates the multiplication terms as subexpressions during the parse of the addition expression, where:

expr \Rightarrow add \Rightarrow mult + add \Rightarrow pri * mult + add \Rightarrow a * mult + add \Rightarrow a * pri + add \Rightarrow a * b + add \Rightarrow a * b + mult + add \Rightarrow a * b + pri * mult + add \Rightarrow a * b + c * mult + add \Rightarrow a * b + c * pri + add \Rightarrow a * b + c * d + add \Rightarrow a * b + c * d + mult \Rightarrow a * b + c * d + pri * mult \Rightarrow a * b + c * d + e * mult \Rightarrow a * b + c * d + e * pri * mult \Rightarrow a * b + c * d + e * f * mult \Rightarrow a * b + c * d + e * f * pri \Rightarrow a * b + c * d + e * f * g.

The derivation tree is shown in Figure 2.2 below.

<code><expr></code>	\rightarrow	<code><add></code>
<code><add></code>	\rightarrow	<code><mult></code> + <code><add></code>
<code><add></code>	\rightarrow	<code><mult></code>
<code><mult></code>	\rightarrow	<code><pri></code> * <code><mult></code>
<code><mult></code>	\rightarrow	<code><pri></code>
<code><pri></code>	\rightarrow	<i>a</i>
<code><pri></code>	\rightarrow	<i>b</i>
<code><pri></code>	\rightarrow	<i>c</i>
<code><pri></code>	\rightarrow	<i>d</i>
<code><pri></code>	\rightarrow	<i>e</i>
<code><pri></code>	\rightarrow	<i>f</i>
<code><pri></code>	\rightarrow	<i>g</i>

Figure 2.1: Operator Precedence Example

For any subtree \mathcal{T}' in \mathcal{T} with root node $u_{\mathcal{T}'}$, the *level* $L(\mathcal{T}')$ is defined to be equal to the length of the path from the root node of tree \mathcal{T} to node $u_{\mathcal{T}'}$. We use the notation $L_{\mathcal{T}'}(x)$ to represent the level of the subtree \mathcal{T}' whose root node is labelled by symbol x .

Definition 11 Let \mathcal{N} be the set of natural numbers and let Σ be a ranked alphabet. The *precedence* of any element $f \in \Sigma$ is defined by a function $p : \Sigma \rightarrow \mathcal{N}$, where $p(f) = 0$ if and only if $f \in \Sigma_0$. For every $n \geq 0$, P_n denotes the subset of Σ consisting of the function symbols of precedence n . We have $P_0 = \Sigma_0$, thus the set of constants all have the same precedence.

2.5 Summary

This chapter has introduced the formal concepts of a language grammar and a derivation tree for a sentence in the language described by the grammar. An algebra appropriate to the problem domain is assumed, and given this algebra, we show in Chapter 3 how expressions in the language can be related to interpretation functions defined for symbols in the algebra.

Chapter 3

Expression Evaluation

“To solve a sticky arithmetical problem, we need a way of setting out the problem which makes it perspicuous. Ordinary arithmetic convention gives us such a way. Two minutes with a pencil on the back of an envelope lets us solve problems which we could not do in our heads if we tried for a hundred years. But at present we have no corresponding way of simplifying design problems for ourselves.”

Notes on the Synthesis of Form, C. Alexander

3.1 Grammar Specification

Let $G = \{N, T, P, \Phi\}$ be an unambiguous context free grammar. This thesis considers a specific grammar for expression formation whose production rules P are of the form:

$$\langle P_1 \rangle \rightarrow \langle P_2 \rangle \langle P_{1'} \rangle \text{ opt} \quad (1)$$

$$\langle P_{1'} \rangle \rightarrow f_1 \langle P_2 \rangle \langle P_{1'} \rangle \text{ opt} \quad (2)$$

$$\langle P_2 \rangle \rightarrow \langle P_3 \rangle \langle P_{2'} \rangle \text{ opt}$$

$$\langle P_{2'} \rangle \rightarrow f_2 \langle P_3 \rangle \langle P_{2'} \rangle \text{ opt}$$

.

.

.

$$\langle P_n \rangle \rightarrow \langle Q_1 \rangle \langle P_{n'} \rangle \text{ opt}$$

$$\langle P_{n'} \rangle \rightarrow f_n \langle Q_1 \rangle \langle P_{n'} \rangle \text{ opt}$$

$$\langle Q_1 \rangle \rightarrow f_{n+1} \langle Q_1 \rangle \quad (3)$$

$$\langle Q_1 \rangle \rightarrow Q_2$$

$$\langle Q_2 \rangle \rightarrow c \quad (4)$$

Figure 3.1: A Generalized Expression Grammar

Production rules denoted (1) and (2) describe the derivation rules for a binary operator f_i . These rules describe a general syntax for a binary expression of the form: $X_1[opX_2[\dots]]$. Rule (3) describes the productions for a unary operator of the form: $[op[op\dots]]X_1$. Rule (4) describes the grammar terminal set, where $c \in T$. The precedence of operators is such that $p(f_i) < p(f_j)$ if and only if $i < j$.

The semantics of this grammar can be defined informally. We consider the case of an operator of rank n . Let S be a sentence of $L(G)$, and let \mathcal{T} be the derivation tree of S with respect to G . The nodes of this tree are labelled by symbols of $N \cup T$. A (usual) production rule of the form $X_0 \rightarrow X_1x_0X_2 \dots X_n$, where $X_0, \dots, X_n \in N$, $x_0 \in T$, will be considered as an element of P of arity n . $\{X_1\}$ is the *left hand side* (LHS) of terminal symbol x_0 , and $\{X_2, \dots, X_n\}$ is the *right hand side* (RHS). Consider a node $u \in \mathcal{T}$ labelled by X_0 with sons u_1, u_2, \dots, u_n . This corresponds

to a production rule of the form described above. Let $\mathcal{A} = (\mathcal{A}, \mathcal{T})$ be an algebra over alphabet $\Sigma \subseteq T$. For algebra \mathcal{A} , every symbol f in Σ_n has an interpretation function $I(f)$, represented as the function $f(\alpha_1, \alpha_2, \dots, \alpha_n)$, where f is understood to be the label of the tree node at u_0 , and α_i is interpreted as the label of the node u_i for all $1 \leq i \leq n$. The sort or type of x_0 is the sort of f . The sort of X_0 is the sort of x_0 . Function $I(f)$ maps to a node $u \in T$ such that the sort of X_i equals the sort of α_i , for $1 \leq i \leq n$.

An example of an abbreviated context free grammar, which incorporates the generalized expression grammar of Figure 3.1, is shown in Figure 3.2. In this example, we isolate a class of operator symbols $\Sigma = \{ +, -, *, /, !, \div \}$ as a subset of the terminal symbol set T . The set Σ is further partitioned into two disjoint subsets O_I and O_D , where $O_I = \{ +, -, *, / \}$ and $O_D = \{ !, \div \}$. Set O_I represents the set of *intrinsic operators* predefined within the programming language. Set O_D represents the set of *derived operators* which are new operator symbols introduced into the language grammar specification. Section 4.2 of this thesis will develop this concept further.

3.2 Sentence Evaluation

Sentence evaluation assigns a symbolic value to any string $S \in L(G)$. Given any string $S \in L(G)$ where G is an unambiguous context free grammar, a unique derivation tree T for S exists. Furthermore, given an algebra \mathcal{A} over an alphabet Σ , for all symbols $f \in \Sigma$ of rank $n \geq 0$, a semantic interpretation function $I(f) : A^n \rightarrow A$ exists. We evaluate a sentence by mapping the algebra of the problem domain onto the language grammar such that any expression in the language can be expressed as a string using the semantic functions of the algebra.

Algorithm 1 specifies an evaluation method for expression grammars of the type described above. The algorithm uses the derivation subtree levels as evaluation levels for tokens in the terminal vocabulary.

```

<expression> → <operation>
<operation> → <unary-op> <operand>
<operation> → <operand> <binary-op> <operand>
<unary-op> → <intrinsic-unary-op>
<unary-op> → <derived-unary-op>
<binary-op> → <intrinsic-binary-op>
<binary-op> → <derived-binary-op>
<operand> → <expression>
<operand> → <identifier>
<operand> → <constant>
<operand> → ( <expression> )
<intrinsic-unary-op> → +
<intrinsic-unary-op> → -
<intrinsic-binary-op> → +
<intrinsic-binary-op> → -
<intrinsic-binary-op> → *
<intrinsic-binary-op> → /
<derived-unary-op> → !
<derived-binary-op> → ÷
.
.
.

```

Figure 3.2: An Abbreviated Expression Grammar Example

Algorithm 1 An evaluation method for expressions in this grammar is:

1. For all $x \in S$, assign a level $l(x) = \min(L_{T'}(x))$ such that for all subtrees T' of T , x is on the leftmost path from the root of T' .
2. For all $x \in S$ where $n = |S|$, assign an ordering x_1, x_2, \dots, x_n representing a left to right traversal of the leaf nodes in T .
3. Set the value of a unique attribute $Q(x) = F$ for all $x \in S$.
4. The value $v(S)$ is a string constructed by processing the elements of S in order, as follows:
 - (i) For all $x_i \in S$, $1 \leq i \leq n$ such that $Q(x_i) = F$ do:
If x_i not in Σ , then set $v(x_i) = x_i$, next i
 - (ii) If x_i has a left hand side operand, then for $j = i - 1$ to 1 ,
Find the first j such that $Q(x_j) = F$ and $l(x_j) \leq l(x_i)$
or, if no such j exists, set $j = 1$.
Let $LHS = \{x_j, x_{j+1}, \dots, x_{i-1}\}$
 - (iii) If x_i has a right hand side operand, then for $k = i + 2$ to n ,
Find the first k such that $Q(x_k) = F$ and $l(x_k) \leq l(x_{i+1})$
or, if no such k exists, set $k = n + 1$.
Let $RHS = \{x_{i+1}, x_{i+2}, \dots, x_{k-1}\}$
 - (iv) If $LHS = \emptyset$ and $RHS = \emptyset$, let $v(x_i) = I(x_i)$
If $LHS = \emptyset$ and $RHS \neq \emptyset$, let $v(x_i) = x_i(v(RHS))$
If $LHS \neq \emptyset$ and $RHS = \emptyset$, let $v(x_i) = x_i(v(LHS))$
If $LHS \neq \emptyset$ and $RHS \neq \emptyset$, let $v(x_i) = x_i(v(LHS), v(RHS))$
Where x_i had a left hand side operand, set $l(x_i) = l(x_j)$
For all $x \in LHS \cup RHS$, set $Q(x) = T$, next i
5. Let $v(S) = \{v(x) : x \in S \text{ and } Q(x) = F\}$

Figure 3.3: Algorithm 1 - Expression Evaluation

Given a derivation tree for a sentence $S \in L(G)$ and a set $\Sigma \in T$ of operator symbols, such that for all $f \in \Sigma$ a semantic interpretation function $I(f)$ exists, Algorithm 1 converts each symbol f into its interpretation function $I(f) = f(\alpha_1, \alpha_2, \dots, \alpha_n)$, where n equals the rank of f . The algorithm traverses the tree in left to right order, examining each leaf node of the tree. For each operator symbol f , at tree node u_f , the algorithm uses the derivation tree to establish the symbolic value for all terms $\alpha_1, \dots, \alpha_n$, $0 \leq i \leq n$.

The grammar production rules used to derive S are assumed to follow the form specified in Figure 3.1. The derivation tree therefore (usually) contains the symbolic value for term α_1 to the left of the node u_f . Values for terms $\alpha_2, \dots, \alpha_n$ are found to the right of node u_f .

The characteristics of f describe the method for constructing the interpretation function $I(f)$. If f has rank 0, then f is a constant in Σ , and $I(f)$ is interpreted as an element $I(c)$ in the sort or carrier for f . If f has rank 1, then f may be either a prefix or postfix function. If f is a prefix function, then the symbolic value for α_1 is found to the right of node u_f . If f is a postfix function, then the value for α_1 has been determined and is found to the left of node u_f . If f has rank greater than 1, then the (usual) construction of $I(f)$ will obtain the value of α_1 from the left of node u_f , and recurse to obtain the value of all terms $\alpha_2, \dots, \alpha_n$ from the right of node u_f .

When an operator symbol f has been evaluated, then the value $I(f)$ represents the subtree for $f(\alpha_1, \dots, \alpha_n)$. Sentence evaluation continues from the leaf node in T following the evaluation of f .

For example, using the grammar of Figure 2.1, if $\Sigma = \{*, +\}$, the evaluation levels for the expression terminal symbols are given in Table 3.1. These values are derived from the derivation tree shown in Figure 2.2. We examine all subtrees in which the symbol occurs on the leftmost path from the root node of the subtree. The level assigned to a symbol is the minimum level of all subtree root nodes examined.

Symbols x	a	*	b	+	c	*	d	+	e	*	f	*	g
Level $l(x)$	1	4	4	3	3	5	5	4	4	6	6	7	7

Table 3.1: Evaluation Levels

The evaluation of S proceeds from left to right. The first operator symbol in Σ evaluates to the string $*(a,b)$ where a is the LHS of $*$ and b is the RHS. The symbol a is identified as the LHS of $*$ because $l(a) \leq l(*)$. The symbol b is identified as the RHS of $*$ because $l(b) = 4$, and $l(+)$ is not $\leq l(b)$. The evaluation of the operator $*$ results in $l(*)$ being set to $l(a) = 1$.

The next operator symbol in the expression is $+$ whose LHS is the string $*(a,b)$, previously evaluated. The right hand side is the symbol set $\{c, *, d, +, e, *, f, *, g\}$. The first symbol of the RHS set is c , with $l(c) = 3$. The RHS terminates at the end of S because all the remaining symbols in S have a level greater than $l(c)$.

The symbol set $\{c, *, d, +, e, *, f, *, g\}$ is recursively evaluated as follows. The first operator symbol in this set is $*$, with $l(*) = 5$. The value of this symbol is the string $*(c,d)$, calculated as described above. The level $l(*)$ is set to $l(c) = 3$. The next operator symbol is $+$, whose LHS has the value $*(c,d)$. The RHS is the symbol set $\{e, *, f, *, g\}$. This set is recursively evaluated.

The recursive value of $\{e, *, f, *, g\}$ is the string $*(e,*(f,g))$. Unfolding all recursions, the symbolic value of S is the string $+(*(a,b),+(*(c,d),*(e,*(f,g))))$.

Chapter 4

Operator Overloading

“How queer everything is today! ... Let me think: was I the same when I got up this morning? I almost think I can remember feeling a little different. But if I’m not the same, the next question is, Who in the world am I? Ah, *that’s* the great puzzle!”

Alice in Wonderland, Lewis Carroll

4.1 Operator Overloading in C++

In Chapter 2, an *operator* was introduced as a terminal symbol in the language alphabet of rank > 0 . The semantic interpretation of this symbol was defined through an interpretation function, which mapped the various sorts or data types of the symbol arguments to the symbol. The set of sorts in the domain and range of the interpretation function is called the *signature* of the operator symbol.

A characteristic of almost every programming language is that it is possible for several operations to share the same name or symbol [16]. This is called *overloading*. For example, the operator symbol `+` typically denotes (at least) two operations, one with signature `integer × integer → integer`, and one with signature `real × real → real`.

In many programming languages, overloading is restricted by:

1. only intrinsic operations are overloaded.
2. for every overloaded operation, the type of the result may be deduced from the name of the operation and the types of the operands, without reference to the context of the expression.

For example, in a language where the `+` operator was overloaded as above, the operation `2 + 3` would be identified as `integer × integer → integer` regardless of the context.

Simple overloading, as described above, is available in C. The C++ language extends the capabilities of C to permit the declaration of a set of operations which can be performed on nonprimitive objects represented as a C++ class [6, 22]. Defining operators to operate on class objects sometimes allows a programmer to provide a more conventional and convenient notation for manipulating class objects than could be achieved using only the basic functional notation. However, C++ fails to address either of the overloading restrictions noted above due to the following problems: [6]

1. C++ cannot be extended by inventing new operator symbols. For example, an “exponentiation” operator cannot be created using the characters `**`. These characters do not form a legal operator in C or C++. Operator overloading is restricted to the existing operator symbols available within the language.
2. The number of operands which an operator takes cannot be changed. For example, the logical NOT operator (`~`) is a unary operator and it cannot be overloaded to be a binary operator. The expression `a = ~ b` is legal, whereas `a = b ~ c` is illegal.
3. An operator’s precedence cannot be changed. The multiplication operator has a higher precedence than the addition operator, so multiplication is performed first when the expression `a = b + c * d` is evaluated. The `*` and `+` operators cannot be overloaded in such a way that the addition is performed first. Parenthesis must be used to alter the order of evaluation. One consequence of this is that the operator chosen for a particular purpose may not have the precedence appropriate for the intended meaning. For example, the `^` operator may seem an appropriate choice for exponentiation, but its precedence is lower than that of addition, possibly introducing some confusion in the expression notation.
4. An operator’s associativity cannot be changed. When an operand is between two operators which have the same precedence, it is grouped with one or the other depending on its associativity. For example, addition and subtraction are both left associative, so the expression `a = b + c - d` is evaluated as `a = (b + c) - d`. The `+` and `-` operators cannot be overloaded in such a way that the subtraction is performed first. Parenthesis must be used to control evaluation order.
5. The meaning of an operator cannot be changed with respect to intrinsic data types. For example, the `+` operator cannot be overloaded for `int` data types.
6. Overloaded operator functions that differ only in return type may not have the same name. Two function declarations of the same name refer to the same

function if they have identical argument types. For example, it would not be possible to overload the division operator to provide versions for both integer and floating point return types, for identical integer arguments.

7. The following operators in C++ cannot be overloaded.

Operator	Description
.	Class member operator
.*	Pointer to member operator
::	Scope resolution operator
?:	Conditional expression operator

Generalized operator overloading is an attack on problems 1, 2, 3, and 5. The proposed solution to these problems introduces a modifiable grammar specification in which new operator symbols can be declared and mapped to semantic interpretation functions in the problem domain. However, a solution to problem 6 requires that an overloaded operator be understood in context with the expression in which it appears. This violates the premise that we have a context free grammar as discussed in this thesis. Problem 4, operator associativity, is a characteristic of the language grammar, however this thesis does not consider this a significant problem as it can be easily resolved through the use of parenthesis. The restrictions imposed by problem 7 suggest that there are different classes of language operators which can be overloaded; these are discussed below.

4.2 Operator Classification

We classify language operators into *intrinsic operators* and *derived operators*. Intrinsic operators are the standard operators predefined within the language specification. Examples of intrinsic C operator symbols are:

`+, -, &, ~, <<, >>, ^, =, +=, ->, ., ++, (), [], ...`

The intrinsic operators are further subdivided into a class of resolution operators and functional operators. *Resolution operators* are operators used by the compiler for unique identifier name qualification, scope resolution, and identifier initialization. These include the first three operators noted in problem 7, plus the initialization operator =, the indexing operator [], and the function call operator (). Resolution operators provide language functions for the declaration and maintenance of program entities or variables and therefore do not fall within the scope of expression evaluation. *Functional operators* are all other operators which are used for expression evaluation. The functional operators are candidates for generalized operator overloading through the definition of new semantic interpretation functions.

When a functional operator is redefined, it becomes a derived operator. *Derived operators* represent the new operations introduced into the language. An example of a derived operator for C is the APL vector sum reduction operator, +/, which uses the functional + operation to sum individual elements of a vector. The semantic interpretation functions for a derived operator are, at their lowest level, implemented with language functional operators.

4.2.1 Assignment and Initialization

The notion of assignment is fundamental to any programming language. The semantics of any assignment $x \leftarrow y$ require that the value of y be stored in x . If an assignment operator is overloaded, then its implementation function must access x by reference. A *reference* is an alias, or alternate name for a variable or object. The main use of references is in specifying arguments and return values for functions in general, and for overloaded operators in particular. The semantics of a reference are similar to that of a pointer, in that any use of the reference variable results in an access to the object to which the reference applies. For example, if the assignment $x \leftarrow y$ is defined by the function $Assign(x, y)$, then x must be passed as a reference parameter to the $Assign$ function so that the function can access the storage location to which x

refers. When x is itself a function, then the function must return a reference which is synonymous with the value of x . This requirement shows that host language support for reference variables is a necessary condition for generalized operator overloading, whenever assignment type operators are derived.

Initialization is different from assignment. Initialization is a compiler activity to preset initial values for program variables. This is a resolution operator, as discussed above, and therefore not applicable to expression evaluation.

4.2.2 Indexing and Name Qualification

Both name indexing and name qualification serve to refine the specific instance or occurrence of an identifier within a program. For example, in C the term $V[3]$ uses the index $[]$ operator as a function to isolate a specific occurrence of an item in vector V . Similarly, the term `John.name` uses the period $(.)$ operator to uniquely qualify the variable `name` to the structure `John`. Indexing is an offset to a named identifier, whereas name qualification clarifies an ambiguous identifier name. Operators which qualify identifier names are resolution operators. Index operators are offset functions, which may or may not be resolution operators depending on the characteristics of the host language. For example, if array bounds or structure size is a known visible attribute of an identifier, then indexing can be implemented as a function. Conversely, if identifier sizes are known only to the compiler, then indexing must be considered as a resolution operator.

4.3 Function Overloading

Function overloading is the name given to the idea of using the same identifier for two or more different functions. For example, inverting a geometric figure is clearly a different operation from calculating the inverse of a matrix, yet, for a consistent notation, each inverse function should have the same name. The correct function

associated with an identifier is determined through semantics, based on the type of the function operands.

Function overloading is a necessary condition for operator overloading. Each overloaded operator has an associated interpretation function as specified in the algebra of the problem. Function overloading therefore describes a relation between the operator symbol and many different interpretation functions. Let the different interpretation functions for operator symbol f be denoted by the set $\{f', f'', \dots\}$. Semantic information, as defined by the domain and range of the expression operands, must be used to choose the correct implementation function f from the set of functions $\{f', f'', \dots\}$ established for the operator if a context free grammar is used.

For cases where many function implementations are defined over the same domain and range, a conflict will clearly exist. Ambiguities of this type are considered to be an error in the algebraic specification of the problem notation. Conflicting name resolution is dependent on context information outside the scope of the expression and the algebra of the problem domain.

4.4 Semantics of Overloaded Operators

The semantics for a derived operator are the rules which define how the operation is to be evaluated. The evaluation method is a function $f(\alpha_1, \alpha_2, \dots, \alpha_n)$ where n represents the arity of the operator. The algebraic specification of the interpretation function f is dependent on the various sorts, or data types, of the function arguments. For example, the Σ operator may be syntactically defined as a unary operator and considered to be semantically meaningful over vector data types. However, for notational consistency, the operator should apply to any vector data type implicitly or explicitly defined within the program language.

Many programming languages implement operator semantic processes by providing type conversions between language data types. However, where data types are

implemented through objects, type conversion is not obvious.

Function f represents an overloaded operator. We informally define an overloaded operator as a family of functions which are consistent only with respect to their semantics. Each function may have a unique computational method and return a different data type as a result. Unfortunately, the determination of the semantic consistency between functions is a difficult issue and is a research subject [1, 23].

The recognition of the correct syntactic implementation for an overloaded operator necessitates either a compile time selection of the appropriate function, or dynamic linking of the proper module into the code during program execution. This thesis does not discuss techniques for function name resolution. Our approach to this problem is to assume that features exist within the host language for static or dynamic selection of the correct function.

4.5 Operator Modification

To this point, we have implicitly assumed that all derived operators are fully and completely specified as a unique symbol within the language grammar. Yet concise notation often requires that standard operations be modified through appropriate subscripts or additional notation. For example, the summation operator \sum applied to a vector operand implicitly assumes a summation over all elements in the vector. A more precise notation may specify $\sum_{i=1}^n$ to denote specific modifiers for the sum.

Operator modification requires that new arguments be passed to the semantic function. The overloaded operator function f is a function of m arguments, $m \geq n$, the first n representing the operands, and the remainder $m - n$ representing optional parameters. Operator modifiers are specified within the language as optional syntactic terms associated with the operator symbol. Different modifier syntax generations are implemented by defining a set of production rules for the new operator symbol.

The general grammar for expressions whose operator functions can contain modifiers is given below. Each operator symbol is represented with a production rule of the form (1), with an optional list of modifiers, described with production rule (2). For simplicity, it is assumed that modifiers are terminal vocabulary symbols.

$$\begin{aligned}
\langle P_1 \rangle &\rightarrow \langle P_2 \rangle \langle P_{1'} \rangle \text{ opt} \\
\langle P_{1'} \rangle &\rightarrow \langle F_1 \rangle \langle P_2 \rangle \langle P_{1'} \rangle \text{ opt} \\
\langle P_2 \rangle &\rightarrow \langle P_3 \rangle \langle P_{2'} \rangle \text{ opt} \\
\langle P_{2'} \rangle &\rightarrow \langle F_2 \rangle \langle P_3 \rangle \langle P_{2'} \rangle \text{ opt} \\
&\cdot \\
&\cdot \\
&\cdot \\
\langle P_n \rangle &\rightarrow \langle Q_1 \rangle \langle P_{n'} \rangle \text{ opt} \\
\langle P_{n'} \rangle &\rightarrow \langle F_n \rangle \langle Q_1 \rangle \langle P_{n'} \rangle \text{ opt} \\
\langle Q_1 \rangle &\rightarrow \langle F_{n+1} \rangle \langle Q_1 \rangle \\
\langle Q_1 \rangle &\rightarrow Q_2 \\
\langle Q_2 \rangle &\rightarrow c
\end{aligned}$$

$$\begin{aligned}
\langle F_i \rangle &\rightarrow f_i \langle M \rangle \text{ opt} & (1) \\
\langle M \rangle &\rightarrow \langle Q_2 \rangle \langle M \rangle \text{ opt} & (2)
\end{aligned}$$

Figure 4.1: A Generalized Grammar for Operator Modifiers

4.6 Summary

This chapter has discussed some limitations of operator overloading as is currently implemented for the C language. A classification of intrinsic language operators into resolution operators and functional operator is proposed. This classification isolates the operators for expression evaluation from the operators used for the declaration and maintenance of program entities and variables. Overloaded operators are recognized as describing a relation between an operator symbol and many different semantic interpretations. The definition of generalized operator overloading is

constrained to apply the language theory and problem algebra discussed in Chapter 2 to language systems which provide support for reference variables and function name resolution. The general grammar production rules for expression evaluation, introduced in Chapter 3, are extended to support the notion of operator modifiers. Modifiers are additional arguments supplied to the operator semantic interpretation function.

Chapter 5

Implementation Design

“Real world constraints always bastardize the most elegant design. However, people tend to focus on these constraints too soon during the design activity. They miss out on the potential ‘big win’ that is possible through problem domain understanding.”

Sam Adams, Knowledge Systems Corporation

5.1 Conceptual Model

In our view, the goal of generalized operator overloading is to provide a means for programmers to configure their language to support operations appropriate to the problem they are attempting to code. A facility for defining new operators in the language grammar and implementing the operator semantics is required. Subsidiary objectives of the augmented language are:

1. New language operators should be of unary, binary, prefix, and postfix type.
2. New operator definitions should be visible to independently compiled program modules. Once the semantics of a language operator is defined, it should be usable on any system that supports the overloaded operator facility.
3. Writing, testing, and maintaining language operator definitions should be no more difficult than writing application code.
4. It should be possible to redefine existing language operators and apply operators to different data types within the language.

Our conceptual model for implementing generalized operator overloading is based on the notion of modifying the grammar specification of the program source language [19]. The general system architecture used is shown in Figure 5.1.

The generalized operator overloading tool is a language preprocessor which has as input an extended language grammar specification and a program written in the language defined by the grammar. The preprocessor acts as a compiler and translates the extended program source code into a C language program which is subsequently compiled by a standard compiler. The compiler output is linked with an interface library which supplies the semantic implementation functions for the operator functions introduced into the extended language grammar.

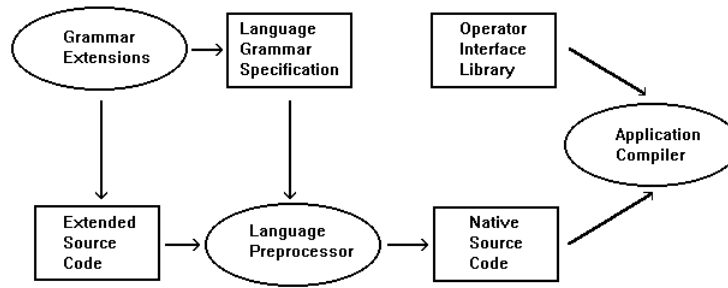


Figure 5.1: Conceptual Model for Generalized Operator Overloading

The preprocessor converts each overloaded operator to a function call appropriate for the operator. Overloaded operators may be of unary or binary type with appropriate modifiers. The function call is constructed by parsing the symbolic expression according to the grammar production rules.

5.2 Grammar Specification

The language grammar specification is a text file which can be edited with any standard text editor. A context free grammar is specified in the file. The language grammar contains production rules for language statements, declarations, and expressions. Appendix I contains an abridged grammar for the C language, and Figure 5.2 illustrates the grammar file syntax.

A grammar production rule set consists of a nonterminal symbol, denoted with a terminating colon (:). Each nonterminal symbol is subsequently followed with a set of derivation rules, each rule consisting of one or more terminal or nonterminal symbols.

All symbols are delimited by spaces. Each derivation rule must be fully specified on one text line. Comments may be used to document logical sections within the grammar file. Comments are lines of text which begin with a period (.) in column 1.

A production rule set must be specified for every nonterminal grammar term. The production rule set may not contain any left recursive generations. Nonterminal generations may be specified as optional terms, denoted with the *opt* keyword. Ambiguous generations, in which one rule is contained within a larger rule, require that the more restrictive rule appears first in the production rule set. This expedites the program parse.

Semantic flags, of the form **S name**, may be attached to grammar production rules. These flags are used to control code generation and parse error recovery. Three flags are defined whose purpose is described below.

```

Declaration Section.
  constant:
    floating-point-constant
    integer-constant
  identifier:
    .
    .
Statement Section.
  statement: S Recover
    jump-statement
    iteration-statement
    selection-statement
    expression-statement
  iteration-statement:
    while ( expression ) statement
    .
    .
Expression Section.
  expression:
    assignment-expression expression opt
  assignment-expression:
    unary-expression assign-operator assignment-expression
    conditional-expression
  unary-expression:
    unary-operator cast-expression
    postfix-expression
  conditional-expression:
    .
    .
Operator Section.
  assign-operator:
    = S 7 Assign
    +=
  unary-operator:
    +
    -
  postfix-operator:
    .
    .

```

Figure 5.2: A Grammar Syntax Specification

Expedite: This directs the parser to accept the current token as a successful derivation for the current grammar production rule. This flag is typically used for `identifiers`, `string literals`, and `integer` or `octal constants`. This flag is used to expedite the program parse, based on the assumption that the lexical analyzer returns syntactically valid tokens.

Recover: This directs the parser to set a code output recovery point. All internal tables and parse history are cleared when an output point is found. This flag is typically used on `statement` production rules.

Modifier: This flag is used to denote a modifier type production rule. Modifier production rules, when invoked during a successful parse of the input, cause a `modifier` semantic flag to be attached to every term generated from the rule. This flag directs the code generation phase to include the terminal tokens as modifiers to an operator function.

5.3 Grammar Extensions

Operator overloading extensions are characterized by new operators being introduced into the expression and operator sections of the language grammar. New operator symbols may be composed as a conjunction of existing language symbols or from new symbols created in the language alphabet. The 7-bit ASCII code set has been reserved for the standard language alphabet. New operator symbols may be allocated in the upper 128 positions of the extended ASCII character set, or through multi-byte character set extensions.

Operator definitions are added to the language grammar by inserting new production rules in the appropriate operator section of the grammar text file. Operator symbols are associated with a semantic evaluation function by attaching a semantic definition, `S n name`, to the symbol. Figure 5.2 shows a semantic evaluation function

attached to the assignment operator symbol.

The function **name** is substituted for the operator in the expression during the code generation phase of the language preprocessor. The parameter n is a numeric flag which defines the operator characteristics as shown in Table 5.1. The numeric flag can be understood in terms of its binary representation. Each bit in the number selects an operator characteristic if the bit is set. Bit 0 represents the least significant bit in the numeric flag.

Bit	Meaning
0	Operator has right hand side operand
1	Operator has left hand side operand
2	Generate function call parameter list
3	Encode function parameters as strings
4	Right hand side call by reference
5	Left hand side call by reference

Table 5.1: Operator Semantic Flags

The flags at bit positions 0 and 1 are interpreted as left and right hand operator characteristics during sentence evaluation as indicated in Algorithm 1. If neither bit 0 or bit 1 is set, then the operator symbol does not have a left hand side or a right hand side operand. The symbolic function **name** is substituted for the operator symbol during code generation. If bit 0 is set and bit 1 is clear, then the operator is a prefix unary operator. Code generation replaces the operator symbol with a function call constructed according to the settings for bits 2 through 5. Similarly, if bit 1 is set and bit 0 is clear, then the operator is a postfix unary operator, and an appropriate function call is constructed during code generation. If both bit 0 and bit 1 are set, then the operator symbol represents a binary function.

If bit position 2 is set, the operator interpretation function is generated as a symbolic function call, using parenthesis around the function parameter list, and a

comma as a list separator. If bit position 3 is set, each function parameter is encoded as a string literal. Bit positions 4 and 5 were introduced in an unsuccessful experiment to emulate reference variables for assignment operators in C.

5.4 Language Preprocessor

The language preprocessor is a one-pass, three step compilation process as shown in Figure 5.3.

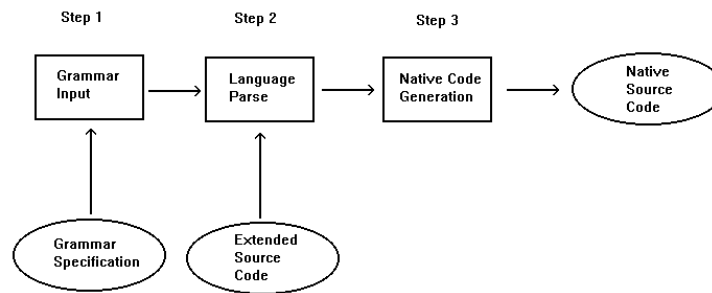


Figure 5.3: Language Preprocessor Architecture

5.4.1 Grammar Input Module

The grammar input module reads the grammar specification for the source language. The grammar rules are parsed and an internal linked list structure is created to represent the language specification. A list of permissible tokens, called the *first set*, is created for each grammar term. The first set lists all tokens which can be generated from the grammar term. This set is used to expedite the language parse. Algorithm 2 is used to create the internal grammar structure.

The algorithm begins by reading the grammar text file, described in Section 5.2. Each nonterminal symbol in the grammar, syntactically recognized by a terminating colon (:), creates a grammar list item. All derivation rules for the grammar nonterminal symbol are parsed. Each rule contains a list of one or more symbols. The rule

lists are linked to the grammar list item.

When all grammar production rules are read, and all nonterminal grammar symbols are known, the first list is built for each grammar term. The construction of the first list is a two phase process. The first phase processes the grammar structure. This phase establishes, for every nonterminal grammar symbol which appears as the first term in some derivation rule, the names of the secondary grammar symbols for which it is a first term.

The second phase reprocesses the grammar structure, this time searching for terminal symbols which appear as the first term in some derivation rule. Each terminal symbol is attached to the first list of the grammar item in which it appears. It is also recursively attached to the first list of all other secondary grammar items in which the current grammar item is known as a first term.

Algorithm 2 Grammar Definition

1. Given an input string S representing the text of a grammar production rule set containing terminal and nonterminal symbols and semantic indicator terms, a function $t(S)$ which returns the next token from S , and a distinct grammar list H_G initialized to null:
2. For all tokens $t = t(S)$ in the grammar source file, do:
 - (a) If t is a nonterminal grammar symbol, create a grammar item G for t , and attach G to the list of known grammar items H_G . Grammar item G contains null lists of the clauses (production rules) associated with G , the first terms derivable from the clauses of G , and a first clause list.
 - (b) If t is a grammar item semantic term, associate t as an attribute of G .
 - (c) If t is the start of a new production rule for G , create a new clause item C for t , and attach C to the list of production rules for G .
 - (d) If t is a new term in clause C , semantic or otherwise, create a new term item T for t and attach T to the current clause item C of G .
3. For all grammar symbols G_i in H_G , do:
 - (a) For all clause items C_j in G_i , set $k = 1$. T_{j_k} is the first element of C_j .
 - (b) If T_{j_k} exists in H_G , create a first clause item F for G_i and attach F to the first clause list of grammar item T_{j_k} . If T_{j_k} is denoted as an optional grammar term, set $k = k + 1$ and repeat Step 3(b).
4. For all grammar symbols G_i in H_G , do:
 - (a) For all clause items C_j in G_i , set $k = 1$. T_{j_k} is the first element of C_j .
 - (b) If T_{j_k} does not exist in H_G , then create a new term FL for T_{j_k} and attach FL to the first term list of G_i .
 - (c) For all F on the first clause list of G_i , do:
 - i. If T_{j_k} is a member of the first list of F , process the next F .
 - ii. Create a new term FL for T_{j_k} and attach FL to the first term list of F .
 - iii. Recurse on Step 4(c) for $G_i = F$. Then process the next F .
 - (d) If T_{j_k} exists in H_G and T_{j_k} is denoted as an optional grammar term, set $k = k + 1$ and go to Step 4(b).

Figure 5.4: Algorithm 2 - Grammar Construction

5.4.2 Lexical Analysis Module

The language parser is a top-down recursive descent parser. A lexical analyzer is used to return tokens to the parser. The lexical analyzer, as implemented, is specific for C. It returns tokens delimited by white space or non-alphanumeric characters. C and C++ comment strings are ignored. Character constants, string constants, and numeric literals are returned as complete tokens. Algorithm 3 is used for lexical analysis.

Algorithm 3 Lexical Analyzer

1. Given an input string S of length n , set i equal to the number of consecutive non-alphanumeric characters beginning in S .
2. If $i = 0$, then set j to the index of the first non-alphanumeric character in S and return the substring $\{S_1, \dots, S_{j-1}\}$ as a token.
3. Otherwise, set j equal to the number of leading white space characters in S . Adjust S to be the remaining characters in S . Let $S = \{S_{j+1}, \dots, S_n\}$. If $j = i$, go to Step 1.
4. If S begins a comment of length k , adjust S to be the string $\{S_{k+1}, \dots, S_n\}$ and set i equal to the number of consecutive non-alphanumeric characters in S .
5. If S begins a string literal, return the string literal as a token.
6. If S begins a numeric constant, return the numeric constant as a token.
7. Set j equal to the index of the next white space or alphanumeric character in S . If $j < i$, then return the substring $\{S_1, \dots, S_{j-1}\}$ as a token. Otherwise, go to Step 1.

Figure 5.5: Algorithm 3 - Lexical Analysis

5.4.3 Language Parse Module

The recursive descent parser identifies feasible grammar production rules for the current input token based on the first set. No token lookahead is performed. The first production rule matched for a grammar term is accepted as an appropriate parse of the current input string. If no rules match then a parse error exists. Parsing recovers at the next appropriate recovery point which is defined as the end of a language statement or translation unit. The path through the parse tree is maintained as tokens are successfully parsed. Each path node is assigned a level based on the recursive descent position within the parse tree. This level is used during code generation to isolate expression terms associated with language operators. Algorithm 4 is used for parsing.

The parse path PP generated with Algorithm 4 is input to the code generation phase. Semantic terms attached to the path entries are used to generate the appropriate function calls for the operator symbols in the language source, using the semantic flags from Table 5.1. Expression operands are isolated using the appropriate parse path level indicators. Algorithm 1 in Chapter 3 describes the method used for sentence evaluation. Figure 5.7 shows the parse path generated for the expression $x = (a + b) \div c$ using the complete C grammar in Appendix I.

5.5 Operator Interface Library

The operator interface library contains the application program interface for the grammar extensions. It represents the mapping of the problem notation to the problem implementation. Object class libraries are an appropriate vehicle for this interface due to the polymorphic nature of object services, where the same service can be defined for many different object types. Figure 5.8 illustrates a class library implementation for addition of integer vectors and simple integer numbers.

Algorithm 4 Recursive Parsing

1. Given an input string S representing the program to be parsed, a grammar $G = \{N, T, P, \alpha\}$ containing a set of production rules P with a start symbol or goal α in P , a function $t(S)$ which returns the next token from S , a flag $f \rightarrow \{T, F\}$ indicating success or failure of the parse, and an integer L representing the parse recursion level:
 2. Increment the parse level L .
 3. If $t(S)$ is not an element of the first set for α , return (F).
 4. Create a parse path entry PP for $t(S)$, with attribute $l(t(S)) = L$.
 5. For all productions p in P derived from α , such that $\alpha \rightarrow \beta_{p_1}\beta_{p_2} \cdots \beta_{p_k}$, do:
 - (a) For all terms $\beta_{p_i} \in \beta_p$, $1 \leq i \leq k$, do:
 - i. If β_{p_i} is a semantic term, then associate the term with the current PP entry, and continue with the next i .
 - ii. If $\beta_{p_i} \in T$ and $\beta_{p_i} = t(S)$, then continue with the next i .
If $\beta_{p_i} \in T$ and $\beta_{p_i} \neq t(S)$, then go to Step 5(b).
 - iii. If $\beta_{p_i} \in N$ then set f equal to the success or fail of the recursive parse of S , beginning at Step 2, for goal β_{p_i} .
 - iv. If the parse succeeds, then continue with the next i .
If $f = F$ and β_{p_i} is an optional term, then do the next i .
If $f = F$ then go to Step 5(b).
 - (b) If all $\beta_{p_i} \in \beta_p$ have been parsed, go to Step 6.
Otherwise, repeat Step 5 for the next production p in P .
 6. If a production p in P was completely parsed, return (T).
Otherwise, return (F).

Figure 5.6: Algorithm 4 - Recursive Parsing

Level	Grammar Term	Symbolic Code
1	translation-unit	main(){x = (a+b)÷c;}
6	direct-declarator2	()
4	compound-statement	{ x = (a + b) ÷ c; }
5	statement-list	x = (a + b) ÷ c;
10	assignment-operator	=
10	assignment-expression	(a + b) ÷ c
13	expression	a + b
16	additive-expression2	+ b
13	expression (LHS)	a
17	multiplicative-expression (RHS)	b
12	multiplicative-expression2	÷ c
10	assignment-expression (LHS)	(a + b)
13	cast-expression (RHS)	c

Figure 5.7: Parse Path Example

The operator interface library may also embed a non-object structured system. In this case, generalized operator overloading uses the library as an application program interface, which maps a new operator symbol to an external system function.

```

class Vector
{ // Definition of an Integer Vector class.
public:
    Vector( int N );           // Constructor
    ~Vector();                // Destructor
    int *v;                    // Vector pointer
    int n;                     // Size of vector
};

int Sum (int a, int b)
{ // Function to add two integers.
    return(a + b);
}

Vector *Sum (Vector *v1, Vector *v2)
{ // Function to add two integer vectors.
    int i;
    Vector *v3;
    v3 = new Vector(v1->n);
    for (i = 0; i < v1->n; i++)
        v3->v[i] = v1->v[i] + v2->v[i];
    return (v3);
}

```

Figure 5.8: Class Interface Library Example

Chapter 6

Results

“Taking Three as the subject to reason about -
A convenient number to state -
We add Seven, and Ten, and then multiply out
By One Thousand diminished by Eight.

“The result we proceed to divide, as you see,
By Nine Hundred and Ninety and Two:
Then subtract Seventeen, and the answer must be
Exactly and perfectly true.

The Hunting of the Snark, Charles Dodgson

Four results of operator overloading are described below. In all cases, we introduce new operators to the C language, with the use of C++, for convenience only, for final compilation. An 8 bit extended ASCII character font is used to encode the new operator symbols introduced into the language.

6.1 APL Functions in C

The first example introduces APL operators to the C language. Vector operators for vector creation, reversal, summation, and patterns are shown in Table 6.1. The program in Figure 6.2 evaluates an APL derivation of the statement:

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} \quad (6.1)$$

Symbol	Name	Use	Function Description
ι	<i>integer</i>	ιn	Create a vector of the first n integers
$+/$	<i>sum reduction</i>	$+/ v$	Sums of the elements of a vector
\emptyset	<i>reversal</i>	$\emptyset v$	Reverses the elements of a vector
ρ	<i>pattern</i>	ρm	Creates a vector of n patterns of m
\div	<i>division</i>	$x \div y$	Division function
\times	<i>times</i>	$x \times y$	Product function

Table 6.1: APL Operator Symbols

The grammar extensions introduced to the C language are shown in Table 6.2.

unary-operator:	mult-operator:
+ / \mathcal{S} 5 VecSum	\div \mathcal{S} 7 Divide
ι \mathcal{S} 5 CreateVec	\times \mathcal{S} 7 Multiply
\emptyset \mathcal{S} 5 ReverseVec	ρ \mathcal{S} 7 PatternVec
assignment-operator:	add-operator:
=	+ \mathcal{S} 7 Sum
<<	

Table 6.2: APL Operator Extensions for C

The grammar used for the expression parse is shown in Figure 6.1. Note that the grammar defines the operator precedence as:

unary > mult > add > assignment

```

expression:
  unary-expression assignment-operator expression
  additive-expression
additive-expression:
  multiplicative-expression additive-expression2 opt
additive-expression2:
  add-operator multiplicative-expression additive-expression2 opt
multiplicative-expression:
  unary-expression multiplicative-expression2 opt
multiplicative-expression2:
  mult-operator unary-expression multiplicative-expression2 opt
unary-expression:
  unary-operator unary-expression
  primary-expression
primary-expression:
  ( expression )
  identifier
  constant

```

Figure 6.1: A Grammar for APL Expressions in C

APL Grammar Extensions for C

```
/* A test of the statement:  $+/\iota N \leftrightarrow ((N+1) \times N) \div 2$  */  
  
void main()  
{  
    int N = 5;  
  
    cout <<  $+/\iota N$ ;  
    cout <<  $+/\phi \iota N$ ;  
    cout <<  $((+/\iota N) + (+/\phi \iota N)) \div 2$ ;  
    cout <<  $(+/( (\iota N) + (\phi \iota N) )) \div 2$ ;  
    cout <<  $(+/( (N+1) \rho N )) \div 2$ ;  
    cout <<  $((N+1) \times N) \div 2$ ;  
}
```

Preprocessor Output Code

```
void main()  
{  
    int N = 5;  
  
    cout << VecSum(CreateVec(N));  
    cout << VecSum(ReverseVec(CreateVec(N)));  
    cout << Divide((Sum((VecSum(CreateVec(N))),  
        (VecSum(ReverseVec(CreateVec(N)))))),2);  
    cout << Divide((VecSum((Sum((CreateVec(N)),  
        (ReverseVec(CreateVec(N))))))),2);  
    cout << Divide((VecSum((PatternVec((Sum(N,1),N))))),2);  
    cout << Divide((Multiply((Sum(N,1),N)),2);  
}
```

Figure 6.2: APL Operators in C

6.2 Logic Operators in C

This example illustrates the definition and use of operators for inferencing and hypothesis evaluation in a logic system. We assume that an external logic system exists for maintaining a knowledge base and inferring new facts. The operator interface library acts as the application program interface to the logic system. The logic operator operands are passed as symbolic entities to the external system for evaluation.

The grammar production rules for the hypothesis operators introduce the notion of an expression modifier list as shown in Figure 6.3. These production rules are extensions to the expression and operator sections of the language grammar (Figure 5.2). Semantic flags, attached to the modifier list and operator symbol, cause the operator terms to be passed as string literal parameters in the generated function call.

The example in Figure 6.4 shows the assertion of facts regarding items on a restaurant menu and hypothesis evaluation regarding the types of meals offered in the restaurant.

```
expression-statement:
    hypothesis-operator ;
    postfix-expression assertion-operator ;
    expression opt ;

expression-modifier:  S Modifier
    constant-expression expression-modifier-list opt
expression-modifier-list:
    constant-expression expression-modifier-list opt

assertion-operator:
    → S 14 Assert expression-modifier opt
hypothesis-operator:
    H S 13 Hypothesis expression-modifier
```

Figure 6.3: Logic Operator Grammar Rules

Note that in this example, the hypothesis and assertion operators have been introduced into the language as expression statements. This restricts the use of the operators to a specific statement type. The operators do not participate in a normal expression derivation, and therefore cannot be used in a compound expression.

Logic Operators in C

```
void main()
{
    hors_d_oeuvres(Artichauts_Melanie) → ;
    meats(Grillade_de_boeuf) → ;
    fishes(Bar_aux_algues) → ;
    desserts(Melon_en_surprise) → ;
    maincourses(m) → meats(m) ;
    maincourses(m) → fishes(m) ;
    meals(h,m,d) → hors_d_oeuvres(h) maincourses(m)
        desserts(d) ;

     $\mathcal{H}$  maincourses(m) ;
     $\mathcal{H}$  meals(h,m,d) fishes(m) ;
}
```

Preprocessor Output Code

```
void main()
{
    Assert("hors_d_oeuvres(Artichauts_Melanie)") ;
    Assert("meats(Grillade_de_boeuf)") ;
    Assert("fishes(Bar_aux_algues)") ;
    Assert("desserts(Melon_en_surprise)") ;
    Assert("maincourses(m)", "meats(m)") ;
    Assert("maincourses(m)", "fishes(m)") ;
    Assert("meals(h,m,d)", "hors_d_oeuvres(h)", "maincourses(m)",
        "desserts(d)") ;

    Hypothesis("maincourses(m)") ;
    Hypothesis("meals(h,m,d)", "fishes(m)") ;
}
```

Figure 6.4: Logic Operators in C

6.3 An Improved C

This example redefines the standard C operators to alleviate some of the common programming errors within the language. In this case, each new operator symbol is implemented as a simple text substitution and does not require an associated operator interface library for expression evaluation. We redefine the standard assignment operator, equality operator, and logical operators as shown in Table 6.3, and provide an example program coded with the new operators as shown in Figure 6.5.

C Operator	New Symbol	Purpose
=	←	Assignment
==	=	Equality Comparison
&	$\bar{\wedge}$	Bitwise AND
	$\bar{\vee}$	Bitwise OR
&&	\wedge	Logical AND
	\vee	Logical OR

Table 6.3: C Operator Redefinition

The grammar adjustments made to the operator section of the C language are shown in Table 6.4. A semantic flag value of zero results in a text substitution of the original operator symbol for the new symbol.

assignment-operator: $\leftarrow S 0 =$	and-operator: $\vec{\wedge} S 0 \&$
equality-operator: $= S 0 ==$	inclusive-or-operator: $\vec{\vee} S 0 $
logical-and-operator: $\wedge S 0 \&\&$	logical-or-operator: $\vee S 0 $

Table 6.4: C Operator Semantic Interpretation Definition

C Operator Extensions

```
void main()
{
    int a, b, c;
    a ← b ← c;
    if ( (a = b) ∧ (b = c) ∨ (a = c) )
        c ← a  $\vec{\wedge}$  b  $\vec{\vee}$  c;
}
```

Preprocessor Output Code

```
void main()
{
    int a, b, c;
    a = b = c;
    if ( (a == b) && (b == c) || (a == c) )
        c = a & b | c;
}
```

Figure 6.5: New C Operators

6.4 The Poetry of C

Our last example uses generalized operator overloading to compile and execute the poem which began this chapter. Compiling poetry is a curious application of the technique presented in this thesis. Figure 6.6 shows the source poem, followed by the preprocessor output code.

Every word (except *true*) in the poem is a member of a ranked alphabet Σ . The numeric words *One*, *Two*, *Three*, *Seven*, *Eight*, *Nine*, *Ten*, and *Ninety* are all constants of rank 0. Their semantic interpretation is their numeric value. The words *Hundred* and *Thousand* are postfix operators of rank 1. They apply to the preceding term, and their semantic interpretation is to multiply the term by the appropriate power of 10. The words *add*, *and*, *multiply*, *divide*, *diminish*, and *subtract* are all symbols of equal precedence, of rank 2. These functions perform the mathematics. The words *By* and the symbol $:$ are constants of rank 0 whose values are the symbols (and) respectively. These words are used in the grammar to define the syntax of an expression group. The word *must* is overloaded as an assignment operator of the form $a \rightarrow b$; this assigns the result of the poem to the variable *true*. All other words in the poem are constants of rank 0 whose semantic interpretation is the null string.

The word *and* is used in this poem in two different contexts, one as an arithmetic operator, and the other as a phrase conjunction. A context free grammar cannot determine the difference, therefore we distinguish between the two uses of the word by encoding the conjunction form as the concatenation of two unique tokens; the *a* which is a special character different from the usual character *a*, and the pair *nd*.

The result of compiling the poem is a valid expression in C.

```

#include <stdio.h>

#define Sum(a,b) a + b
#define Subtract(a,b) a - b
#define Multiply(a,b) a * b
#define Divide(a,b) a / b
#define Exp2(a) a * 100
#define Exp3(a) a * 1000

void Assign(int a, int *b) { *b = a; }

/*****
/*
/*          An evaluation of a poem.
/*
/*
/*****/

void main()
{
    int true;

        Taking Three as the subject to reason about
        A convenient number to state
    We add Seven, and Ten, and then multiply out
        By One Thousand diminished by Eight:

        The result we proceed to divide, as you see,
        By Nine Hundred and Ninety and Two:
    Then subtract Seventeen, and the answer must be
        Exactly and perfectly true ;

    printf("The exact and perfect answer is %d \n",true);
}

```

Figure 6.6: A Poetic C Program

Preprocessor Output Code

```
#include <stdio.h>

#define Sum(a,b) a + b
#define Subtract(a,b) a - b
#define Multiply(a,b) a * b
#define Divide(a,b) a / b
#define Exp2(a) a * 100
#define Exp3(a) a * 1000

void Assign(int a, int *b) { *b = a; }

/*****
/*
/*          An evaluation of a poem.
/*
/*
*****/

void main()
{
    int true;

    Assign(Subtract(Divide(Multiply(Sum(Sum(3,7),10),
        (Subtract(Exp3(1),8))), (Sum(Sum(Exp2(9),90),2))),17),&true) ;

    printf("The exact and perfect answer is %d \n",true);
}
```

Chapter 7

Conclusions

The conclusion reached here may be stated in terms of two schoolboy definitions for salt. One definition is, “Salt is what, if you spill a cupful into the soup, spoils the soup.” The other definition is, “Salt is what spoils your soup when you don’t have any in it.”

A History of Mathematical Notations, Florian Cajori

In this thesis we have provided a method for implementing problem based notation within a programming language. Our method for extending a language grammar uses operator overloading as a technique to effectively map object services to new language operator symbols. This technique is shown to be feasible for the case where the language is defined through a context free grammar.

The notion of a derived operator was introduced in Section 4.2. A derived operator represents the class of language operators defined for the problem and introduced into the language. The derived operators specify a problem specific notation which is syntactically compatible with existing language operators. Derived operators can also be applied to the standard language to redefine existing operator syntax and semantics.

Our experience with this method has shown that it is an effective means for modifying the language structure, providing that unambiguous operator symbols are used. When derived operators are created from existing operator symbols, ambiguous code generation can occur. We resolve this by using graphic symbols for problem notation rather than representing symbols through various key combinations from a standard keyboard.

Operator overloading is predominantly a static syntactic facility for language extension. The scope of any new operator definition applies throughout the program module or translation unit, as specified by the augmented grammar. A complete application can be constructed from many modules, each compiled using different algebras and semantic interpretation functions.

Language support for reference variables was found to be a necessary condition for the synonymous treatment of identifier values and function values as function operands.

The use of a modified language grammar eliminates many of the operator overloading restrictions in the C++ language associated with operator definition, precedence, associativity, and meaning. We were successful in introducing domain specific

problem notation into the C language.

The results of our work confirm that language definitions can be adapted to fit the problem notation. Our model uses a programmable grammar for expression and operator syntax, where the operator semantics are implemented through object services or functions defined for the problem. Operator overloading is not new, but the idea of using it for problem notation is new.

Current limitations of this work, and areas for future research are:

- Our current preprocessor does not interpret or process `include` files specified in the C source program. This limitation can cause syntax errors during the parse, because not all program identifiers are known during the compilation process. This problem frequently manifests itself when user data types are defined externally and referenced in the source program.
- Research into methods for performing semantic function name resolution can extend our generalized operator overloading method. Function name resolution is used to select the correct function definition, in cases where the same name is given to two or more functions. Section 4.4 of this thesis discusses this topic. Currently, we assume that features of the host language perform function name resolution.
- Our model for generalized operator overloading applies to derived operators which are based on the functional operators intrinsic to the host language. Further research is necessary to extend the model to incorporate the language resolution operators, introduced in Section 4.2 of this thesis.
- Generalized operator overloading is defined for unambiguous context free grammars. New research, directed towards extending our model to support context sensitive grammars, would extend the set of languages applicable to generalized operator overloading.

- Our current solution for generalized operator overloading does not qualify or confirm the validity of any grammar extensions introduced into the host language. This can lead to ambiguities in the augmented grammar. The problem is most frequently noted when new operator symbols are constructed from existing language operator symbols. Research into automated methods for grammar verification may resolve this problem.
- Our implementation of generalized operator overloading uses an extended character set in which new operator symbols are defined. The operator symbol is considered a unique entity in the language vocabulary. Research into methods for decomposing an operator graphic symbol into its component parts may provide an improved algebraic notation for operator symbols which accept modifier terms.

Bibliography

- [1] S. Antoy and J. Gannon. Using Term Rewriting to Verify Software. *IEEE Transactions on Software Engineering*, pages 259–274, April 1994.
- [2] P. Boullier. Syntax Analysis and Error Recovery. In *Methods and Tools for Compiler Construction*, pages 7–44. Cambridge University Press, 1983.
- [3] F. Cajori. *A History of Mathematical Notation*. Open Court Publishing Co., 1928.
- [4] P. Coad and E. Yourdon. *Object-Oriented Design*. Prentice Hall, Inc., 1991.
- [5] L. Cordelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17, 1985.
- [6] Microsoft Corporation. *Visual C++ Tutorial*. Microsoft Corporation, 1993.
- [7] B. Courcelle. Attribute Grammars : Definition, Analysis of Dependencies, Proof Methods. In *Methods and Tools for Compiler Construction*, pages 81–102. Cambridge University Press, 1984.
- [8] C. N. Fisher and R. J. LeBlanc Jr. *Crafting a Compiler*. Benjamin Cummings, 1988.
- [9] J. H. Gallier. *Logic for Computer Science*. Harper and Row, 1986.
- [10] M. A. Harrison. *Introduction to Formal Language Theory*. Addison Wesley, 1978.
- [11] K. E. Iverson. *A Programming Language*. John Wiley and Sons, New York, 1962.
- [12] K. E. Iverson. Notation as a Tool of Thought. *Communications of the ACM*, 23:444–465, 1979.
- [13] L. F. Johnson. Thesis redirection. Technical Memo to W. Miles, 1994.
- [14] M. Kilian. Trellis: Turning Designs into Programs. *Communications of the ACM*, 33, 1990.

- [15] T. Korson and J. D. McGregor. Understanding Object-Oriented: A Unifying Paradigm. *Communications of the ACM*, 33, 1990.
- [16] B. Lorha. *Methods and Tools for Compiler Construction*. Cambridge University Press, 1984.
- [17] D. B. McIntyre. Language as an Intellectual Tool: from Hieroglyphics to APL. *IBM Systems Journal*, 30, 1991.
- [18] B. Meyer. *Object Oriented Software Construction*. Prentice Hall, 1988.
- [19] W. S. Miles and L. F. Johnson. Implementing Generalized Operator Overloading. Submitted for Publication, *Software Practice and Experience*, 1994.
- [20] B. A. Myers and M. B. Rosson. Survey on User Interface Programming. *ACM CHI '92*, 1992.
- [21] A. Snyder. Encapsulation and Inheritance in Object Oriented Programming Languages. *Communications of the ACM*, 1986.
- [22] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 2nd edition, 1991.
- [23] R. D. Tennent. *Mathematical Semantics and Design of Programming Languages*. PhD thesis, University of Toronto, 1973.
- [24] D. A. Watt. Contextual Constraints. In *Methods and Tools for Compiler Construction*, pages 45–80. Cambridge University Press, 1983.
- [25] R. Wirfs-Brock and R. Johnson. Survey in Current Research in Object-Oriented Design. *Communications of the ACM*, 33, 1990.

Appendix I

Grammar for C

This appendix lists the grammar specification for the C language. For brevity, we have omitted the grammar for identifier declarations, to more clearly focus on the expression, operator, and statement sections of the language. An ellipsis notation (...) is used to denote grammar production rules which extend over one line of text.

Preprocessing Section.

```
translation-unit: S Recover
    external-declaration translation-unit2 opt
translation-unit2:
    external-declaration translation-unit2 opt
external-declaration:
    function-declaration
    declaration
    control-line
function-declaration:
    declaration-specifiers opt declarator declaration-list opt ...
    compound-statement
recovery-unit:
    ; recovery-unit
    } translation-unit
    } recovery-unit opt
statement recovery-unit
```

Operator Section.

```
unary-operator:
    &
    *
    +
    -
```

~

!

assignment-operator:

=

*=

/=

%=

+=

-=

<<=

>>=

&=

^=

|=

postfix-operator:

++

--

[expression]

(expression opt)

. identifier

-> identifier

multiplicative-operator:

*

/

%

additive-operator:

+

-

shift-operator:

<<

>>

relational-operator:

<=

>=

<

>

equality-operator:

==

!=

and-operator:

&

exclusive-or-operator:

^

inclusive-or-operator:

|

logical-and-operator:

&&

logical-or-operator:

||

Expression Section.

primary-expression:

(expression)

identifier

constant

string-literal

expression:

assignment-expression expression2 opt

expression2:

```

    , assignment-expression expression2 opt
constant-expression:
    conditional-expression
assignment-expression:
    unary-expression assignment-operator assignment-expression
    conditional-expression
conditional-expression:
    logical-or-expression ? expression : conditional-expression
    logical-or-expression
postfix-expression:
    primary-expression postfix-expression2 opt
postfix-expression2:
    postfix-operator postfix-expression2 opt
unary-expression:
    ++ unary-expression
    -- unary-expression
    sizeof ( type-name )
    sizeof unary-expression
    unary-operator cast-expression
    postfix-expression
cast-expression:
    ( type-name ) cast-expression
    unary-expression
multiplicative-expression:
    cast-expression multiplicative-expression2 opt
multiplicative-expression2:
    multiplicative-operator cast-expression ...
    multiplicative-expression2 opt
additive-expression:
    multiplicative-expression additive-expression2 opt

```

additive-expression2:
 additive-operator multiplicative-expression ...
 additive-expression2 opt

shift-expression:
 additive-expression shift-expression2 opt

shift-expression2:
 shift-operator additive-expression shift-expression2 opt

relational-expression:
 shift-expression relational-expression2 opt

relational-expression2:
 relational-operator shift-expression relational-expression2 opt

equality-expression:
 relational-expression equality-expression2 opt

equality-expression2:
 equality-operator relational-expression equality-expression2 opt

and-expression:
 equality-expression and-expression2 opt

and-expression2:
 and-operator equality-expression and-expression2 opt

exclusive-or-expression:
 and-expression exclusive-or-expression2 opt

exclusive-or-expression2:
 exclusive-or-operator and-expression ...
 exclusive-or-expression2 opt

inclusive-or-expression:
 exclusive-or-expression inclusive-or-expression2 opt

inclusive-or-expression2:
 inclusive-or-operator exclusive-or-expression ...
 inclusive-or-expression2 opt

logical-and-expression:

```

    inclusive-or-expression logical-and-expression2 opt
logical-and-expression2:
    logical-and-operator inclusive-or-expression
        logical-and-expression2 opt
logical-or-expression:
    logical-and-expression logical-or-expression2 opt
logical-or-expression2:
    logical-or-operator logical-and-expression ...
        logical-or-expression2 opt

```

Statement Section.

```

statement: S Recover
    jump-statement
    iteration-statement
    selection-statement
    labelled-statement
    compound-statement
    expression-statement
jump-statement:
    goto identifier ;
    continue ;
    break ;
    return expression opt ;
compound-statement:
    { declaration-list opt statement-list opt }
declaration-list:
    declaration declaration-list2 opt
declaration-list2:
    declaration declaration-list2 opt
statement-list:

```

```
    statement statement-list2 opt
statement-list2:
    statement statement-list2 opt
expression-statement:
    expression opt ;
iteration-statement:
    while ( expression ) statement
    do statement while ( expression ) ;
    for ( expression opt ; expression opt ; expression opt ) ...
        statement
selection-statement:
    if ( expression ) statement else statement
    if ( expression ) statement
    switch ( expression ) statement
labelled-statement:
    identifier : statement
    case constant-expression : statement
    default : statement
```

Appendix II

Font Table

The following table illustrates the extended ASCII character set used to specify mathematical symbols within the C language.

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	10	11
12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f	20	21	22	23
														!	"	#	
24	25	26	27	28	29	2a	2b	2c	2d	2e	2f	30	31	32	33	34	35
§	®	&	'	()	*	+	,	-	.	/	0	1	2	3	4	5
6	7	8	9	:	;	<	=	>	?	@	A	B	C	D	E	F	G
48	49	4a	4b	4c	4d	4e	4f	50	51	52	53	54	55	56	57	58	59
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y
5a	5b	5c	5d	5e	5f	60	61	62	63	64	65	66	67	68	69	6a	6b
Z	[\]	^	_	`	a	b	c	d	e	f	g	h	i	j	k
6c	6d	6e	6f	70	71	72	73	74	75	76	77	78	79	7a	7b	7c	7d
l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}
7e	7f	80	81	82	83	84	85	86	87	88	89	8a	8b	8c	8d	8e	8f
~		≅	À	B	X	Δ	E	Φ	Γ	H	I	⊗	K	Λ	M	N	O
90	91	92	93	94	95	96	97	98	99	9a	9b	9c	9d	9e	9f	a0	a1
Π	⊗	P	Σ	T	Y	ς	Ω	Ξ	Ψ	Z	[∴		⊥	—	—	α
a2	a3	a4	a5	a6	a7	a8	a9	aa	ab	ac	ad	ae	af	b0	b1	b2	b3
β	χ	δ	ε	φ	γ	η	ι	φ	κ	λ	μ	ν	ο	π	θ	ρ	σ
b4	b5	b6	b7	b8	b9	ba	bb	bc	bd	be	bf	c0	c1	c2	c3	c4	c5
τ	υ	ω	ω	ξ	ψ	ζ	≤	∞	f	↔	←	↑	→	↓	±	≥	×
c6	c7	c8	c9	ca	cb	cc	cd	ce	cf	d0	d1	d2	d3	d4	d5	d6	d7
∝	÷	≠	≡	≈	...	ℕ	ℤ	ℝ	∅	⊗	⊕	⊗	∩	∪	⊃	⊇	⊂
d8	d9	da	db	dc	dd	de	df	e0	e1	e2	e3	e4	e5	e6	e7	e8	e9
⊂	⊆	∈	∉	∠	∏	√	·	¬	∧	∨	⇔	⇐	⇑	⇒	⇓	Σ	Γ
ea	eb	ec	ed	ee	ef	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	fa	fb
⌈	⌊	∫	⌋	{		}	~	∀	∃	»	«	×	÷	∅	§		
fc	fd	fe	ff	100	101	102	103	104	105	106	107	108	109	10a	10b	10c	10d
①	②	③	④														