Authoring Tools for Intensional Markup

by

Xing Jin
B.Eng, Beijing Polytechnic University, 1996

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Authoring Tools for Intensional Markup

by

Xing Jin
B.Eng, Beijing Polytechnic University, 1996

Supervisory Committee

Dr. W. W. Wadge, (Department of Computer Science)
Supervisor

Dr. M. Van Emden, (Department of Computer Science)
Departmental Member

Dr. U. Stege, (Department of Computer Science)
Departmental Member

Dr. H. Cazes, (Department of French)
Outside Member

Dr. M. Levy, (Apple Computer Inc.)
External Examiner

Supervisory Committee

Dr. W. W. Wadge, (Department of Computer Science)
Supervisor

Dr. M. Van Emden, (Department of Computer Science)
Departmental Member

Dr. U. Stege, (Department of Computer Science)
Departmental Member

Dr. H. Cazes, (Department of French)
Outside Member

Dr. M. Levy, (Apple Computer Inc.)
External Examiner

ABSTRACT

Several tools have been developed for the authoring of intensional (context-sensitive) documents; for example, IHTML (Intensional HTML), IML (Intensional Markup Language), and ISE (Intensional Sequential Evaluator). However, at present, it is still very difficult to author intensional markup documents. To ease this difficulty, this thesis presents two new intensional authoring tools, IMP (Intensional Macro Processor) and ICC (Intensional C Compiler). IMP is a powerful yet easy to use macro processor embedded with a JavaScript engine. ICC in turn is an intensional extension to ANSI C. Several applications of IMP and ICC such as WIMPAS, intensional spreadsheet and CGI programming in ICC are discussed in this thesis to illustrate how IMP and ICC facilitate intensional authoring.

Table of Contents

List of Tables

List of Figures

Acknowledgement

First of all, I would like to thank my supervisor, Dr. Bill Wadge, for his strong support, great patience and valuable guidance throughout my study and research at the University of Victoria, without which this work would not have been possible.

I would like to thank my committee members, Dr. M. Van Emden, Dr. U. Stege, Dr. H. Cazes and Dr. M. Levy, for the gracious offering of their time, continued attention, and thoughtful advice to my thesis.

In addition, I need to thank Qin Zhu, Paul Swoboda, Elena Li and others who gave me their ongoing creative suggestions.

I would also like to thank my family for their understanding and encouragement during my graduate study at Victoria.

Victoria is one of the most beautiful cities I have had the pleasure of living. I spent much time walking by the ocean, watching the birds, and breathing fresh air. The wonderful nature offered a creative energy and inspiration for life while instilling, no doubt, many of the thoughts in this thesis.

Chapter 1

Introduction

Markup refers to the use of a markup language to describe the structure and appearance of a particular document. Interpreted by a program (such as a web browser or a word processor) or by a compiler (such as LaTeX) into a more readable version of the text, the markup language is typically a description language consisting of delimited characters and codes that are added to the content of the document to represent its structure. The structural information is stored in the same file as the content in a markup document, whereas, traditionally, it is stored in separate files. Intensional markup refers to the use of an intensional language to describe the intensional structure of a particular document (such as an HTML document). Intension and markup add additional levels of information on top of the content of the document. For example, in IML [1], a typical intensional markup language, the content is a first level of information showing in browsers while the HTML markup is a second level providing browsers the rendering information about the content. An intensional language (ISE [2]) is a third level specifying the intensional structure about the content while a macro language (Groff [3]) is a fourth level aiding the intensional markup authoring.

One main problem that arises in the current practice of intensional markup authoring is that even with the availability of powerful tools such as IML and ISE, authoring can still be difficult. Let us look at two examples to illustrate this difficulty and how this thesis chose to address this problem.

The first example writes a simple loop macro in IML, which uses GNU's Groff strictly as a macro processor. The following Groff macro duplicates its first argument three times.

```
.de dup                                          Line 1
.    nr a 0                                      Line 2
.    while (\\na < 3) \{\                        Line 3
\\$1                                             Line 4
.        nr a +1                                 Line 5
.    \}                                          Line 6
..                                               Line 7
```

Line 1 defines a macro named 'dup' while line 7 ends the macro definition. Groff has characteristically strict rules, e.g., a period ('.') must appear as the first character of each line of a macro definition. Line 2 sets register 'a' to zero, while line 3-6 defines a loop. For each loop, line 4 outputs the first argument and line 5 increases register 'a' by 1. Line 3 instructs the loop to exit when register 'a' is no longer less than '3'. Thus a macro request '.dup *' will produce '* * *'.

Now let us note what happens when one nests 'dup' macros, in an attempt to produce nine '*' in a row, like this:

```
.dup "\.dup *"                                   Line 8
```

The inner '.dup *' produces '* * *', which in turn serves as an argument to the outer 'dup'. Although this appears sound, when the above code was input to Groff, one only found three '*' in a row instead of nine as expected. Here, the root cause is the register 'a' which serves as a global variable. Both the inner and the outer 'dup' use the same register 'a'. After being increased up to 3, this register 'a' is never reset to 0, thus only three '*' are shown instead of nine. This is a fatal flaw that Groff cannot be overcome because Groff does not provide any local variable facilities. Without the help of local variables, it is extremely difficult to design a macro works for nested calls. It is quite hard to imagine how a droptext macro (a droptext is a typical example of intensional concept: having two different renderings, depending on the value of the parameter controlling the presence/absence of the droppable text) could be written in IML that allows droptext inside droptext.

The absence of local variables is only one of the strict restrictions imposed by Groff for writing IML scripts. Strict and arcane syntax, limited control structures (Groff only

provides '.if' and '.while' structures), limited data types (Groff only has integer and string types), and limited string manipulation facilities, all contribute to the difficulties of writing IML scripts.

A conclusion of the example shown here is that the macro processing facilities provided by Groff on which IML is based, are probably sufficient to serve as a text formatter - the main reason for its creation and existence, but is assuredly not appropriate for intensional markup authors to write IML scripts. Groff is not a particularly good choice to serve as a general-purpose macro processing tool.

To overcome the problem, it is necessary to find better ways to author intensional markup in a versatile but easy manner. One approach to address this problem is to find a more powerful yet easy to use macro processor to replace GNU's Groff. A survey of existing macro processors did not locate a suitable one. Instead we propose IMP, a new one of our own design.

In IMP, the previous example would be written as the following:

```
$dup = $(                                          Line 1
    var b = bargv(0);                              Line 2
    var r='';                                      Line 3
    for (var i=0; i<bargv(1); i++) {               Line 4
        r+=b;                                      Line 5
    }                                              Line 6
    r;                                             Line 7
$)                                                 Line 8
$dup( $dup($[*$],$[3$]), $[3$] )                   Line 9
```

Line 1 defines a macro named 'dup'. '$(' and '$)' encloses several JavaScript statements as the body of the macro. Line 2 sets variable 'b' with the first argument as its value, while line 4-6 performs a loop. For each loop, 'b' is appended to 'r'. Line 7 outputs the 'r' as the result of the macro expansion. Line 9 calls 'dup' nestedly to produce nine '*' in a row.

As one discovers, intensional markup authors may now benefit from a more free writing style in conjunction with harnessing the powerful programming facilities of the most popular scripting language. With a simple free-style syntax, the macro offers a greater clarity. Using embedded Javascript, intensional markup authors can now write far more

complex macros. In addition, with a built-in IMP document object model (DOM), intensional markup authors can even manipulate the whole text output buffer via scripting, similar to what web authors do inside browsers with the HTML DOM.

To further free the hands of intensional markup authors, WIMPAS is also proposed in this thesis. With the help of WIMPAS, intensional markup authors now only need to launch a web browser to write IMP scripts with the generated ISE page produced almost instantly. One does not require a knowledge of Unix, Vi or FTP, ISE or IMP commands, nor a Unix terminal. With a greater number of predefined IMP macros added into WIMPAS, intensional markup authors will no longer need to know the details of ISE. Writing intensional markup could eventually require several mouse clicks and a few forms to fill out on the web. Figure 1.1 is a screen snapshot of WIMPAS.

Now, let us look at the second example. In the other extreme situation, authors can write ISE scripts directly without any assistance from intermediate macro processing. The following ISE code is an ISE function taken from the 'intra32.ise' - the French Sentences Maker by Bill and Christine Wadge [4].

```
function print_cell()
{
    $tab = []{"tab"};
    $cell = []{"cell"};

    vswitch {
        <[]{tab} . ":" . []{cell}> {
            %"<td bgcolor=white align=center>"%;
            %"<font color=black>"%;
            print_ccont?[cstatus:yes]();
            %"</font></td></icase>"%;
        }
        <> {
            %"<td bgcolor=red align=center>"%;
            print("<a href=\"intra32.ise<",
                    [[]{tab} . ":" . []{cell} . ",
                    tab:-,cell:-"], ">\">");
            %"<font color=white>"%;
            print_ccont?[cstatus:]();
            %"</font></a></td>"%;
        }
    };
}
```

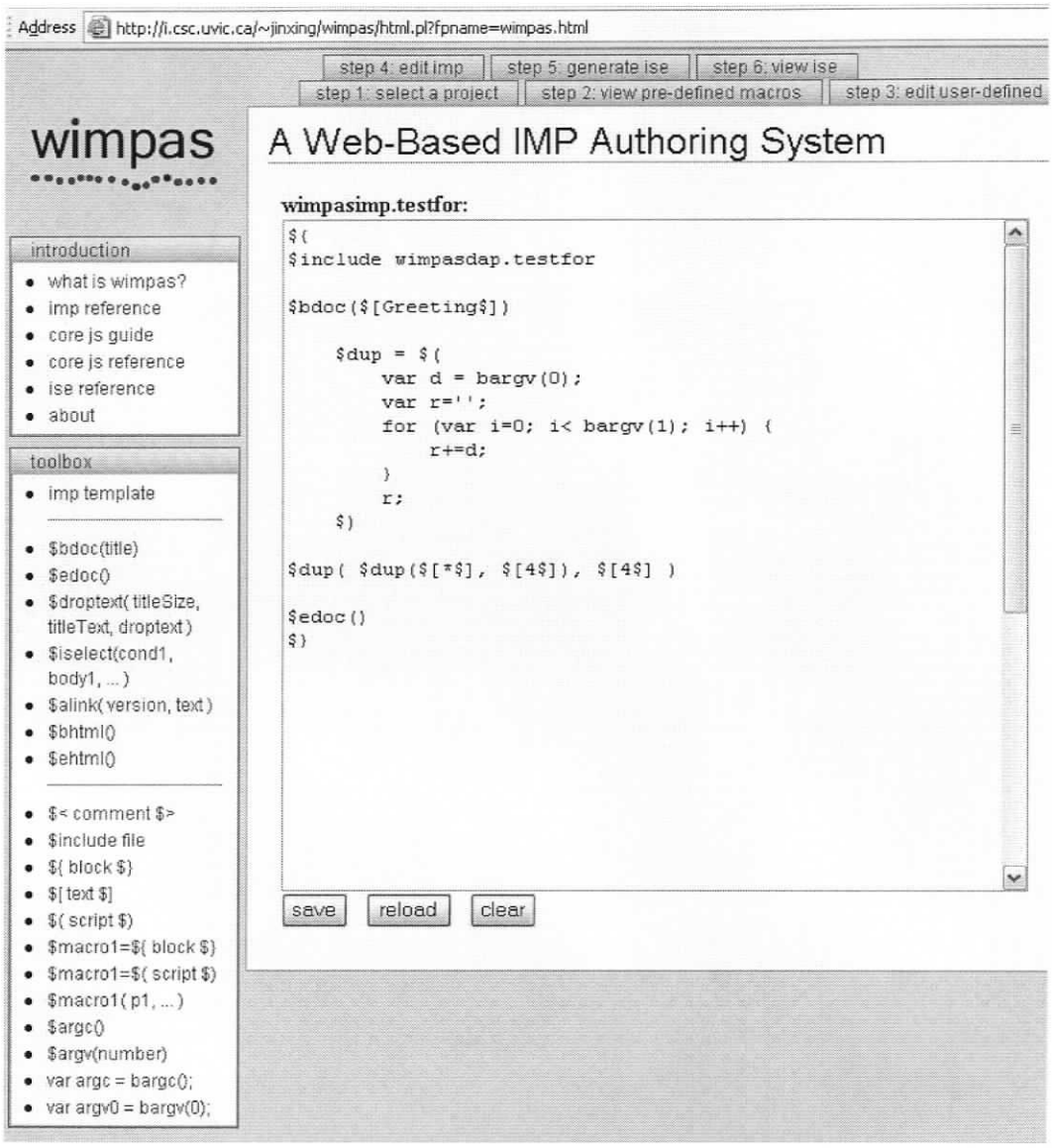


Figure 1.1. *WIMPAS*

In the above code segment, $tab takes the dimension value of dimension 'tab' and $cell takes the dimension value of dimension 'cell'. Based on current context, 'vswitch' either outputs a white color cell filled with black color words or outputs a red color cell filled with hyper-linked white color words.

There is nothing wrong with the ISE code here, but rather the ISE syntax (essentially that of Perl) can be a challenge to understand or write. ISE is a fully featured programming language, borrowing many good features from Perl. However, ISE also inherits Perl's arcane syntax and in actuality makes it even worse. Programming in ISE is usually considered as error-prone and difficult to debug.

(The most recent intensional markup scheme replaces ISE with Perl itself, augmented with an intensional library, which is even more challenging).

One approach to address this problem is to make a relatively simple and easy to understand authoring tool to share the responsibility of ISE. An intensional C compiler, ICC, is proposed in this thesis. Let us see how ICC deals with this 'print_cell()':

```
void  print_cell()
{
    char * x  = _icc_vgetdimval(_icc_rt_version, "tab");
    char * y  = _icc_vgetdimval(_icc_rt_version, "cell");
    char * vmod = mystrcatn(4, x, ":", y, "+tab:+cell:");

    if (*x != 0 && *y != 0 && _icc_vrefinesto(vmod, _icc_rt_version)) {
          printf("<td bgcolor=white align=center><font color=black>");
          vcall(print_ccont, "cstatus:yes");
          printf("</font></td></icase>");
    } else {
          char * v = _icc_vmod(_icc_rt_version, vmod);
          printf("<td bgcolor=red align=center>");
          printf("<a href=\"intra32.cgi<%s>\">", v);
          printf("<font color=white>");
          vcall(print_ccont, "cstatus:");
          printf("</font></a></td>");
          free(v);
    }

    free(x);
    free(y);
    free(vmod);
}
```

Is this a preferred route? Perhaps. If one is more familiar with C, C++ or Java than Perl, one would assume the ICC version of the code as being much better to understand than its ISE brother. However, if one is an expert on Perl, a contrary opinion would ensue.

This thesis discusses the design, implementation, examples and applications of IMP and ICC, with more emphasis given on IMP.

Chapter 2 provides background information on intensional programming, intensional programming languages, intensional markup languages, intensional versioning, macro processors, interpreters and compilers, in addition to offering a brief description of the history and the related works in the areas of intensional markup and intensional programming.

Chapter 3 describes the design and implementation of IMP.

Chapter 4 briefly describes ICC.

Chapter 5 discusses some possible directions for future work that could be taken on intensional programming and intensional markup.

Chapter 6 offers a conclusion for the thesis.

Appendix A includes IMP reference manual that is a complement to Chapter 3.

Appendix B contains ICC reference manual that also details the design and implementation of ICC.

Chapter 2

Background and Related Work

2.1 Intensional Programming

What is Intensional Programming? Bill Wadge wrote in [5]:

> *There is, of course, an easy answer, namely 'programming in a language based on intensional logic'. But this answer raises another, more fundamental question: 'What is intensional logic?' Logicians have been trying to answer that question for about 2500 years...*

Intensional Programming has its foundation in intensional logic, a branch of mathematical logic which was initially proposed and used for expressing the semantics of natural languages. The potential of intensional logic in the design of modern programming language was soon realized and led to the development of an intensional programming paradigm. It is likely that P. Rondogiannis's summary on intensional programming would prove helpful in its understanding.

> *Although it is not easy to give a brief and non-technical explanation of what an intensional language really is, we could say that one of the main characteristics of the paradigm is that it deals with infinite entities of ordinary data values. Such entities could be a stream of numbers, a two-dimensional table of characters, a tree of strings, and so on. These entities are treated as first-class objects by intensional languages: two streams of numbers can be added together as*

> *if they were ordinary data values, functions can be applied on infinite tables and trees, and so on. Due to the above characteristic, intensional languages are especially appropriate for describing the behavior of systems that change with time or physical phenomena that depend on more than one parameters (such as time, space, temperature, etc). Moreover, as intensional languages are based on solid mathematical foundations (i.e. intensional logic), they promote a purer and more declarative way of programming than traditional imperative programming languages.* [6]

Intensional logic has a long and well established, mathematically rigourous foundation; interested readers can refer to [5] for an introduction on this subject. The next section describes several existing intensional programming languages.

2.2 Intensional Programming Languages

In the mid 1970's, Bill Wadge and Ed Ashcroft invented a new kind of programming language, Lucid [7], during their research in program verification. Since then, *"it has been extended and modified several times in the last twenty years. It is with these developments that the very concept of intensional programming was created."* [8] Lucid is a functional dataflow programming language designed to experiment with non-VonNeumann programming models. The simplest dialects of Lucid support only a small set of data types: integers, floats, and symbols, but the most powerful feature of Lucid is the ability to allow programmers to define filters or transformation functions that act on time-varying data streams with the help of a few temporal operators. Noticeably, in Lucid, variables and expressions are streams in a time dimension.

Since its inventation, the original Lucid has evolved into many dialects, such as Indexical Lucid, fLUCID, GLU and Lustre, to name a few. One of them is GLU - Granular Lucid [9] by R. Jagannathan, E.A. Aschroft and A.A. Faustini. GLU is a coarse-grain dataflow language for programming conventional parallel computers. It is a hybrid programming

model that combines the declarative dataflow language Lucid and the procedural language C. A GLU program consists of two parts: the Lucid part that specifies the program composition and a C portion that imperatively defines various data types and data functions referred to in the Lucid part [10]. The combination of intensional declarative style and imperative style has an influence on the creation of ISE [2]. GLU had found a great range of applications in parallel computing. The GLU compiler is available for several Unix systems. Both GLU and ISE greatly influence the creation of ICC presented in this thesis.

One intensional language that is worth mention is Chronolog, a logic programming language based on a linear-time temporal logic [11]. As Lucid is designed with the declarative programming concept and ISE with the imperative programming concept, Chronolog is designed with the logic programming concept. Chronolog programs resemble standard logic programs, with horn-clause syntax decorated by a few intensional operators. Although Chronolog possesses an important theoretical value, it has not been evaluated in practical applications of significant size [6].

Another area that is incorporated into intensional programming is version control. In 1999, P. Swoboda in [2] presents an imperative interpreted scripting language, ISE, short for 'Intensional Sequential Evaluator'. ISE is based on Perl grammar and semantics combined with versioned programming. ISE has most of the data types and control flow of Perl language. In addition, ISE applies intensional versioning for all of its identifiers, data structures, and control structs. *"The basic idea of ISE is that a thread of execution carries with it a current context (or global version), and every time an identifier is used in an expression, an intensional best-fit is made at run-time, against the current context, to determine which version of the identifier is being considered."* [2] ISE has been utilized to implement IML for intensional web programming. *"ISE is the most recent development in a long line of languages and tools that have put forward the concepts of intensional programming, where programs are understood to execute in an implicit multi-dimensional context, whose individual dimensions can be manipulated explicitly as needed by a program"*[12].

Intensional versioning has found a potential application in internet computing where

several intensional markup languages have been designed based on the idea of intensional versioning. The next section briefly describes some intensional markup languages. Please see Chapter 2.4 for details on intensional versioning.

2.3 Intensional Markup Languages

Intensional markup languages are programming languages designed for web programming based on the idea of intensional programming. Bill Wadge in [13] pointed out:

> *The World-Wide Web is the first large-scale experiment in Intensional Programming and Education. ... It should be clear then, that the Web is exactly an intension. The Web is nothing more or less than an indexed family of pages, the indices being the URLs.*

The first system for intensional markup was IHTML1 [14] which was later redesigned and re-implemented as IHTML2 [15]. In IHTML the implicit context (to which documents may be sensitive) is a "version", basically a collection of parameter settings which specify which of a number of alternative variants of a document is requested. IHTML is an intensional version of HTML, in which many of the HTML elements such as links, images, and file inclusions can be versioned. The browsers make requests to specific versions of documents while the servers make intensional computation on the documents for the versioned requests embedded in URLs and deliver the documents to the browsers. The fundamental intensional computation algorithm behind IHTML is an intensional best-fit algorithm that is adapted later by ISE and also by ICC as presented in this thesis.

IHTML2 was later superseded by ISE. P. Swoboda named his application of ISE on HTML as ITHML3. But, in fact, ISE is more than a superior IHTML, being a full featured imperative interpreted intensional programming language.

ISE greatly reduces the effort to create an intensional web site, but it is still quite complicated. In [1], Bill Wadge describes the simple markup language IML, or Intensional Markup Language, to solve this problem. IML is actually a macro package written in

GNU's Groff [3] to produce ISE scripts. The ISE scripts in turn are interpreted by an ISE interpreter. IML hides the details of ISE's complicated syntax from the authors. IML has a great practical value that allows a larger scale of practical IHTML web site constructions.

Recently, the emphasis on WWW moved from HTML to XML. In [16], [17], and [18], an intensional XML scheme is presented, namely MXML, short for Multidimensional XML. V.T. Phong in [19] presents another scheme to intensionalize XML, namely IXML, short for Intensional XML. Both MXML and IXML are attempts to model the IHTML scheme to the XML domain. MXML focuses on a document level by extending the syntax of XML to embed context information into XML document, while IXML focuses on the multiversioning infrastructure of a Web based system for handling intensional XML by utilizing XSL and Servlet.

2.4 Intensional Versioning

Intensional versioning, a key feature of intensional markup languages, is a hybrid of traditional version control and intensional programming. This idea is first described by J. Plaice and W. Wadge in [20].

In intensional versioning, versions are not limited to a linear order. Version labels now can contain strings such as 'graphics', and are not limited to numbers such as '3.2.4' in the traditional manner. Sub versions such as 'graphics%bugfix' and joins of versions such as 'japanese+graphics%bugfix+infinite' are also possible. The relationship among all the possible versions forms a partial order instead of the traditional linear order. To approximate from one version to another version, a variant structure principle is proposed in [20] as follows:

> *... a particular version of an entire system is formed by combining the most relevant existing versions of the various components of the system. We call this the variant structure principle it makes precise the idea that components of a given version of the system can be inherited by more refined versions of the*

system.

[20] gives an algebraic version language for intensional versioning. This algebraic version language is firstly adopted in IHTML1 [14], with a few minor additions. A more practical scheme of version language is described in IHTML2 [15] which is adopted by ISE with a few minor modifications.

To illustrate the idea of the intensional version language scheme, a simplified version scheme is presented, as follows:

$$V ::= \varepsilon \mid D : I \mid D : \varepsilon \mid V + V \tag{2.1}$$

$$D ::= T \tag{2.2}$$

$$I ::= T \mid N \tag{2.3}$$

$$N ::= n \mid N.n \tag{2.4}$$

Where V represents version; ε is the *vanilla* version (i.e. the default or the most generic version); D represents dimension; I represents index; T is a token (i.e. a string composed of letter, number, and '_'); and n is a non-negative integer.

In order to make version computation more practical, a canonical form of version is defined to simplify and standardize versions. Some examples of the canonical form transformations are:

$$\text{a:(b:c+d:e)} \implies \text{a:b:c+a:d:e}$$

$$\text{z:m+t:+a:b} \implies \text{a:b+z:m}$$

An operator '*refinement*' denoted as '$\subseteq$' to expose the possible partial order relationship between two versions, is defined as follows:

$$\varepsilon \subseteq V \tag{2.5}$$

$$V \subseteq V \tag{2.6}$$

$$(V_1 \subseteq V_2) \wedge (V_2 \subseteq V_3) \longrightarrow V_1 \subseteq V_3 \tag{2.7}$$

$$V \subseteq V + V' \tag{2.8}$$

$$m \leq n \longrightarrow d : m \subseteq d : n \tag{2.9}$$

Where m and n are positive integers. For example, ε refines to any version, and 'language:french' refines to 'language:french+graphic:high_resolution'.

Applying the operation of refinement, when a version request is received, an intensional versioning system will perform a best-fit algorithm to search for the closest version refining to the requested version among the version space. For example, we have three versions of a HTML file: ε, 'language:french' and 'language:french+graphic:high_resolution'. When version 'language:french+graphic:high_resolution+colorscheme:daylight' is requested, version 'language:french+graphic:high_resolution' is the best-fit version. When version 'language:french+graphic:low_resolution' is requested, version 'language:french' is the best-fit version. When version 'language:english' is requested, version ε is the best-fit version.

The idea behind the refinement and best-fit algorithms is actually a form of computation model based on intensional concepts. Please see Chapter 4 for the details on the implementation on these algorithms in this thesis. Please also refer to [20], [14], and [15] for intensional versioning concepts.

2.5 Macro Processors

Macro languages have been around since the time of assembler languages dominating the computer programming language world. They used to be very simple text replacement facilities to save the tedious repetition of whole pieces of text in different parts of the same program. Macro languages normally consist of macro definitions and macro calls. [21] offers the formal definition of macro definitions and macro calls.

A macro definition usually takes the following form:

```
$MacroDef
Macro Template
Macro Body
```

```
$MacroEnd
```

$MacroDef and $MacroEnd are delimiters to mark the beginning and ending of a macro definition. A macro template is a string of characters denoting the macro name and the formal parameters of the macros separated by special markers such as commas or spaces. A macro body is a string of characters that determines the output at macro expansion time and usually contains formal parameters which will be replaced by actual parameters at expansion time and may also contain some control logic such as conditional and loop codes. These control logic codes also contribute in determining the output of the macro expansion based on the actual parameters and possibly other information.

A macro call is an input string to be matched against a previously defined macro template with the effect of causing the macro processor to produce output texts which replace the macro call. Macro calls may be nested, that is, macro calls may appear in parameters of other macro calls. In some macro systems, recursive macro calls are also allowed.

Some macro processors are designed as special-purpose macro processors to perform certain kind of tasks, such as assembly language macro processors which are simple and efficient to produce assembly codes and C preprocessors which are aware of the hosting programming language C grammar. Some macro processors are designed as general-purpose macro processors in the sense that their application area is wide. Most general-purpose macro processors are designed as string handling macro processors which are particularly efficient at string manipulation.

The IMP macro processor described in Chapter 3 is a general-purpose macro processor but also pays particular attention to special application areas (i.e. embedding other programming languages such as JavaScript, ISE, and HTML).

2.6 Interpreters and Compilers

An interpreter is a program that executes its source code whereas a compiler does not execute its source code but translates it into executable machine code that is output to a file for later execution.

Interpreting code is usually slower than running the compiled code since the interpreter must analyze each statement in the program each time it is executed and then it performs the desired action, whereas the compiled code merely performs the action. Access to variables is also slower in an interpreter because the mapping of identifiers to storage locations must be performed repeatedly at run time rather than at compile time.

A modern compiler is often organized into many phases. Lexical analysis is the first phase that breaks the source file into tokens, which are passed into syntax analysis phase which parses the phrase structure of the program and builds abstract syntax trees. The abstract syntax trees are passed into semantic analysis phase to determine what each phrase means, relate variables to their definitions, and perform type checking. The result of the semantic phase will be passed on to the translation phase which produces intermediate representation trees (IR trees) in a source language independent manner. The IR trees are passed into the optimization phase which performs various optimizations such as instruction selection, control flow analysis, dataflow analysis and register allocation. The final machine code is generated by the code emission phase.

Some interpreters interpret their source code directly while some other interpreters may also use a lexical analyzer and a syntax parser to produce abstract syntax trees for later interpretation. Often, interpreters have a read-evaluate-print loop as their main interpreting body.

ISE, using op-node trees as its internal syntax structure, is an interpreter for intensional programming. The ICC presented in this thesis takes the compiler approach for intensional programming.

Chapter 3

IMP - An Intensional Macro Processor

3.1 Introduction

Before developing IMP, we conducted a survey of historical and currently active popular general-purpose macro processors in the hope of finding one capable of performing complex macro processing yet easy to use to meet the requirement for intensional markup authoring.

GNU's M4 is probably the most popular macro processor distributed along with Unix and Linux. M4 is a powerful general-purpose macro processor exhibiting conditional and loop controls, many built-in text-handling macros, various input control macros and some debugging supports. However, there are also some fatal shortcomings that effectively exclude the M4 from the candidate list for intensional markup authoring. M4 has no variable facilities, making its conditional and loop controls of limited usefullness. M4 does have the ability to recognize macro calls without resorting to any special, prefixed invocation characters. While sometimes useful, this feature often leads to unwanted macro invocations. Although several techniques are provided to inhibit the recognition of unwanted macro calls, this only serves to render M4 syntax more intricate.

There are several variants related to M4 - M1, M3, and M5. M1 is a tiny macro processor, omitting many features of M4 and serving as a simple string replacing facility. M3 [22] is the forerunner of M4, designed by the same author [23] of M4 mainly for the AP-3 minicomputer. M5 is a variant of M4 with limited improvements.

TRAC, GPM, ML/I and STAGE2 could be the most well known general-purpose macro processors before the 1980's. TRAC [24] (Text Reckoning And Compiling), still in use, provides features (innovative in its day) such as distinguished active invocation and neutral invocation controls, variable length parameters, interactive controls and limited arithmetic operations. The interactive feature of TRAC makes it unique among the many macro processors, while the other features of TRAC can be found on many macro processors made later. GPM [25] (General Purpose Macro-generator) was designed at roughly the same time as TRAC, around the 1960's. It shares with TRAC common features such as conditional expressions and recursive functions. Although it is mainly of historical interest, GPM has been an important ancestor of the more powerful M4. ML/I [26] is another ancient design that is famous for its portability. Using a specially designed simple language called LOWL to encode the logic of ML/I, it is designed for transportability among different computers with different architectures and configurations. Regardless of the complexity of the logic in the implementation of ML/I, it is only necessary to map the statements of LOWL into the instruction set of the new host machine in order to run ML/I. This feature is very attractive if the target host machine is short of compilers. In terms of macro processor features, the ML/I is roughly at the same level as other macro processors designed at the same time. The portable idea of ML/I is further applied in a mobile programming system around 1970, which was offered with a STAGE2 macro processor. STAGE2's approach involves text transformations, somewhat similar to the M4. A notable distinction between STAGE2 and M4 is that STAGE2 supports multi-line and nested constructs. Still, STAGE2 is not powerful enough to write very sophisticated applications.

GEMA [27] represents another kind of macro processor that uses pattern matching and replacement. Some notable features of GEMA are its lack of imposing any particular syntax for what a macro call looks like in addition to its not having any built-in assumptions about input language syntax, its ability to recognize patterns spanning multiple lines and handle recursive patterns, such as matching pairs of nested parentheses, and its offering different sets of rules that can be used in different contexts. In this sense, many Unix utilities

such as GREP, SED and AWK can be regarded as macro processors. The approach of pattern matching and replacement is beneficial in the translation of existing documents while having the power to perform notably complicated transformation tasks. However, pattern matching is rather complicated to learn and similarly difficult to program for intensional markup authors.

There are also some very good special-purpose macro-processing systems available. Lisp macros and some of its dialects such D-Expressions and Scheme macros are powerful yet the difficulties in learning Lisp obstruct their application as an easy-to-use general-purpose macro processor. CPP (C PreProcessor) is also a good macro processing tool, but its intrinsic awareness of C grammar prevents it from being recognized as a general-purpose macro processor. Many large software systems provide macro facilities to extend, automate, and customize the host systems. For example, SAS (Statistical Analysis System) offers a very good macro processor facility, with macro language featuring variables, program statements, expressions and functions, nearing what a fully featured programming language can provide. However, these macro processors depend on the operations of the host systems and are sophisticated in syntax, thus rendering them inadequate to serve as a general-purpose macro processor or as an intensional macro processor.

Among the many special-purpose macro processors, the ones specially designed for XML/HTML transformation, such as XSLT, WebCompile, cPIA, HTMLPP and MP4H, are of special interest to this thesis. Instead of being general-purpose macro processors, they are designed specifically to produce HTML or XML documents. XSLT [28] translates XML into a presentation form, which can be considered a macro processing; both cPIA [29] and WebCompile [30] use an XML syntax to specify macro definitions; cPIA uses a simplified form of the Document Object Model as its representation for its internal document parse tree; HTMLPP [31] is a Perl front-end that creates a set of HTML files by breaking one large file into small pieces; MP4H [32] has the intention to incorporate JavaScript into its macro processing engine. Although none of these macro processors is ideal, many of the ideas behind these macro processors informed the design of IMP.

As discussed, it is difficult to find a macro processor that is both powerful and easy-to-use in the requirement of intensional markup authoring. IMP is an attempt to solve this problem through borrowing and combining many ideas from the macro processors discussed.

3.2 Design

3.2.1 Design Goals

In the following we describe the design goals of IMP:

- Simple - IMP should possess a particularly simple syntax to facilitate learning and ease of use, even for beginners. Its lexical structures are ideally to ease for typing. IMP should also be easily configured for syntax-aware editors. Several example configurations of such syntax-aware editors are included in Appendix A.3.
- Powerful - IMP should have the power to perform sophisticated tasks. Although it is hard to define "powerful", IMP should give users the freedom to express ideas in macros. To achieve this, IMP embeds a JavaScript engine and a DOM engine to allow users to write very complex macros. Actually, these features give IMP the power of a real programming language.
- Flexible - IMP should have a flexible architecture to ease future software evolution. To achieve this, the IMP implementation takes a front-end and back-end architecture allowing the flexibility to independently modify or change its front-end or back-end.
- Manageable - IMP should assume that users do indeed pay attention to the formatting of the output. In particular, the macro processor should do its best to keep spaces and tabs in the original text unaltered. Keeping the output readable and manageable is very important in terms of usability, however, this is usually overlooked by many existing macro processors. By using special text delimiters, IMP allows one to copy all spaces and tabs into its output. The users can expect the formatting of the output

in a certain manner.

- Clear - IMP's own lexical properties should not be in any conflict with the embedded language, for example, ISE. Ideally, as a general-purpose macro processor, it is best designed in a way that its own lexical properties can be adjustable to complement any embedded languages. IMP's carefully selected delimiters allow a particular freedom to code in ISE and other popular languages such as HTML and XML. The initial version of IMP does not support user customizable delimiters.
- Identifiable - IMP should provide facilities such as namespaces or scoped macro names in order that large systems can be divided and conquered without worrying about name conflicts. IMP implements scoped macro names.

There are several other design goals such as variable length of parameter lists, file inclusions, and comments that will be covered in the remainder of this chapter.

The followings are no concerns in the initial version of IMP, but they could be taken into consideration in the future if one desires to improve IMP:

- Intensional macro names.
- Macro processing speed.
- Cross platform supports.
- Named parameters.
- Active invocation.
- Multiple diversions.
- User customizable delimiters.

3.2.2 System Components

Most modern software systems use a multi-tier architecture. Modern compilers also take this approach, as for example, GCC, which internally has a front-end and back-end architecture. The front-end parses the compiled language lexically and syntactically, and the

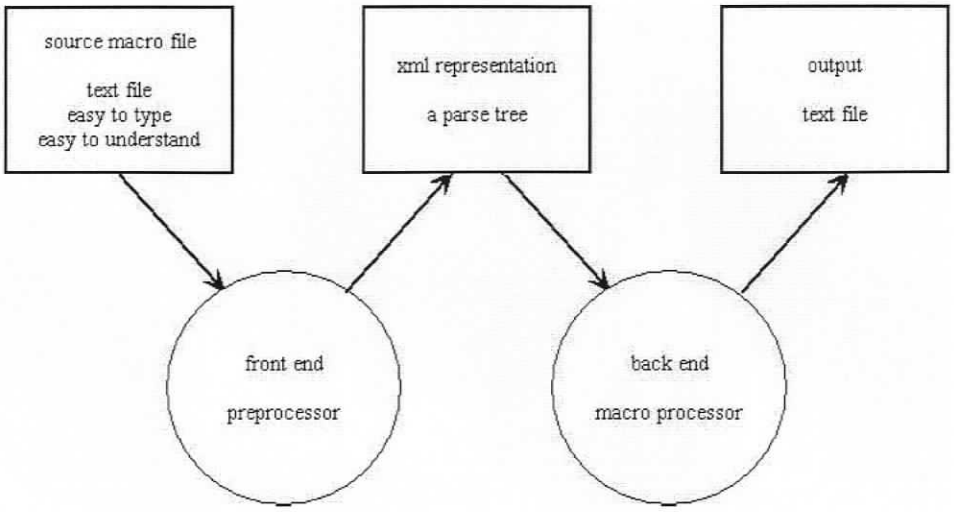


Figure 3.1. *IMP System Components*

resulting syntax tree is passed on to the back-end. The back-end takes the syntax tree, and then generates the compiler output at assembler code level. This architecture gives GCC a high level of reusability. The front-end ensures that GCC is machine independent as much as possible, and the back-end ensures that GCC produces high quality code through intensive optimizations.

IMP follows this approach by taking a front-end and back-end architecture. It is easy to change its front-end to meet various syntax requirements and also easy to change its back-end for efficiency in the future. The benefits of the front-end and back-end architecture are improved usability, maintainability, and flexibility.

Figure 3.1 summarizes the IMP's system components at a particularly high level.

Users write IMP source code files in plain text. The plain text files are fed to the IMP's front-end. The front-end performs the lexical and syntax analysis and either outputs an internal XML representation or reports errors if there are any. The back-end takes the XML representation, performs the actual macro expansion, and outputs the final result.

The interface between the front-end and the back-end is an XML representation. An XML interface is chosen simply because XML is a well-known standard. The interface

Table 3.1. *IMP Language Quick Reference*

Element	Example
Comments	$< ... $>
File Inclusions	$include fileName
Blocks	${ ... $}
Texts	$[... $]
Inline Scripts	$(... $)
Macro Definitions	$macroName = ${ ... $}
Macro Definitions (Script Macros)	$macroName = $(... $)
Macro Invocations	$macroName(param0, param1, param2, ...)
Number of Arguments	$argc()
Number of Arguments (Script Macros)	var argc = bargc();
Arguments	$argv(0)
Arguments (Script Macros)	var argv0 = bargv(0);

can be easily adopted later by anyone who desires to replace IMP's front-end or back-end. It would not be surprising if future compilers also take XML as their internal syntax tree representation, in addition to macro processors. The XML interface will be detailed in Chapter 3.2.5.

3.2.3 Simple Front-End

The following table gives an overview of the IMP front-end language, which is described in subsequent sections.

3.2.3.1 Delimiters

The frond-end is designed by taking simplicity as its primary concern. The design also identifies its possible applications and prioritizes clarity and manageability. Embedding ISE codes and HTML/XML codes are two immediately visible applications. In addition, IMP also has JavaScript embedded. With the existing various macro processors, the delimiter methods vary with different processors but most macro processors use single special characters to delimit the macro definitions and macro calls. For example, Groff uses a '.' at the beginning of a line to start a request line in which macro calls and macro definitions are placed. M1 uses '@' to start its built-in keywords as in the example '@define m1 value' which defines a macro named 'm1'. To allow the special delimiters in text, escapes must be used. For example, in Groff, to begin a text line starting with a '.' control character, an escape sequence '&' must be inserted before the '.'

IMP could also take this single character delimiter approach, but that would put a heavy burden on the user side in that there are many chances that the user would have to pay particular attention to escape every appearance of the special delimiter characters inside the text part, making the IMP source difficult to read and maintain. This becomes more serious when writing macros that embed script codes (e.g. ISE) that in turn embed other script codes (e.g. HTML).

IMP uses $< and $> to denote comments, $(and $) to denote embedded JavaScript, ${ and $} to denote blocks and $[and $] to denote texts. Delimiters such as $<, $>, $(, $), ${, $}, $[and $] are chosen simply because these character sequences rarely appear in ISE or HTML/XML codes. Utilizing a special character, $, helps to make the IMP source more readable and also to ease scanning.

These delimiters greatly clarify the IMP source. There is little need for users to escape the delimiters in IMP, while there is only a small burden to the users who have to type more characters to write the macro definitions and macro calls. This small extra effort can be saved if the author writes the IMP program in an IMP syntax-aware editor or in the WIMPAS, a web-based IMP authoring system. Appendix A has some syntax-aware editor

examples. Chapter 3.5.2 describes the WIMPAS.

3.2.3.2 Standard Macro Definition

IMP has two kinds of macro definitions - standard macro definitions that often have texts in their macro bodies and script macro definitions that have JavaScript statements in their macro bodies.

The grammar for standard macro definitions is:

```
macro    :=   ID = block

ID       :=   $[a-zA-Z_]([a-zA-Z_] | [0-9])*

block    :=   '${' stmt* '$}'

stmt     :=   argc | argv | ilscript | text | ref | smacro
              | macro | block
```

Here is a simple standard macro definition example:

```
$macro = ${
   '$[' This is a standard macro definition. '$]'
$}
```

Macro template starts with an ID name followed by a '='. An ID starts with '$' and is immediately followed by a macro name. A macro name follows exactly the same rules as that of identifiers in programming languages such as C.

IMP follows the Groff's approach for formal parameters having no explicit specification for formal parameters in macro templates. One benefit is that this allows a variable number of parameters, which is rarely seen in macro processors that have formal parameter specifications in their macro template.

The macro body is a block which is enclosed by '${' and '$}' and consists of a sequence of statements. Statements could be the numbers of arguments, arguments, inline scripts, texts, macro references (macro calls), script macro definitions, standard macro definitions or blocks.

Macro definitions can be nested and have scopes. Scopes will be discussed in Chapter 3.2.3.9.

3.2.3.3 Parameter

The initial version of IMP does not support named parameters, thus there is no formal parameter in its macro template. However, IMP supports implicit variable length parameters.

The grammar for parameter reference is:

```
argc      :=   '$argc' '(' ')'

argv      :=   '$argv' '(' NUMBER ')'

NUMBER    :=   [0-9]+
```

Here is an example of using parameter reference:

```
$macro = ${
    $[There are $] $argc() $[parameter(s) passed into this macro. $]
    $[The first parameter passed into this macro is $] $argv(0)
$}
```

The number of parameters can be examined by the built-in macro '$argc()'. This macro will be replaced by the number of actual parameters at macro expansion time. To reference the parameters, another built-in macro '$argv(i)' is provided where 'i' is the index of the parameter in the parameter list. The first parameter always starts from zero.

For embedded JavaScripts, IMP provides two built-in JavaScript functions to examine the number of parameters and the parameters. They are 'bargc()' and 'bargv(i)'. 'bargc()' returns the number of parameters and 'bargv(i)' returns the *i*th parameter in the parameter list where the 'i' starts from zero.

3.2.3.4 Text

The grammar for text is:

```
text      :=    '$[' TEXT '$]'
```

An simple example of IMP text block is:

```
$[This is a text block.$]
```

Texts are enclosed by '$['and '$]'. IMP treats any sequences of characters as text until it encounters '$]'. To escape '$]', any occurrence of '$]' should be denoted by '$$]'. Since '$]' rarely appears in ISE and HTML code, this approach expects very few escapes in practice thus making the texts clear to read and write for the author. Unlike some other macro processors, IMP does not do any pattern matching and replacement in the text. If there are any requirements to apply pattern matching and replacement operations, IMP users can utilize the powerful and easy-to-use regular expression core object 'RegExp' provided by the embedded JavaScript. The 'RegExp' implements all the regular expression features such as pattern matching, searching, and replacing, which can be found in tools such as AWK and SED. In this way, IMP leads to notably clear text content and maintains the ability to perform complicated regular expression operations.

3.2.3.5 Inline JavaScript

Inline JavaScript and script macros are the two means of embedding JavaScript into IMP. This section describes inline JavaScript while script macros are described in the next section.

The grammar for inline JavaScript is:

```
ilscript :=    '$(' JSCODE '$)'
```

Inline JavaScripts are enclosed by '$(' and '$)'. IMP treats any sequences of characters after '$(' as JavaScript codes until it encounters '$)'. To escape '$)', any occurrence of

'$)' should be replaced by '$$)'. Again, since '$)' is rarely found in JavaScript codes, this approach is efficient in writing inline JavaScript.

As soon as the IMP macro processor finds an inline JavaScript, the JavaScript is executed immediately. The value of the last line of the inline JavaScript codes will serve as the replacement text, as shown in the following example.

```
$[123*456=$]

$(
    var r = 123;          // the first line
    r = r * 456;          // the second line
    r;                    // the last line
$)
```

In the above example, the first and the second line of JavaScript are executed, so r becomes 56088 ($56088 = 123*456$). The last line will emit '56088', the value of r, to the output. The final output text for the aforementioned IMP code is therefore '123*456=56088'.

Placing an empty string as the last line of inline JavaScript codes will result in no output text, as shown in the following example.

```
$[123*456=$]

$(
    var r = 123;          // the first line
    r = r * 456;          // the second line
    "";                   // the last line
$)
```

This approach, interpreting the value of the last line of inline JavaScript as the output text, gives the IMP user considerable computing powers and flexibilities.

3.2.3.6 Script Macro Definition

The grammar for script macro definition is:

```
smacro    :=    ID = ilscript
```

Here is an example of script macro definition:

```
$smacro = $(
    var r = 123;          // the first line
    r = r * 456;          // the second line
    r;                    // the last line
$)
```

The Script Macro template starts with an ID followed by a '=' and ends by an inline JavaScript block. The ID and parameter rules are exactly the same as in standard macro definitions. As in the inline JavaScript, the value of the last line of script macros is interpreted as the output text of the script macros. The difference between script macros and inline JavaScripts is that script macros can be referred by their IDs while inline JavaScript codes cannot be referred.

3.2.3.7 Macro Reference

Although script macros and standard macros are quite different in their definition, they share the same syntax for macro references.

The grammar for macro references is:

```
ref      :=   ID '(' ( | param (',' param)* ) ')'

param    :=   argc | argv | ilscript | text | ref | block
```

Macro calls make no difference between standard macros and script macros. A macro call starts with an ID followed by parameters enclosed in '(' and ')'. To avoid confusion in recognition of the formal parameters, ',' is used to separate parameters, a common approach in macro processors and programming languages.

Parameters can be inline scripts, texts, macro calls and blocks. Using blocks as parameters can bind several statements as a single parameter.

IMP supports recursive macro calls. As in most other macro processors and programming languages that support recursive macro or function calls, IMP provides no guard to

avoid infinite recursive macro calls. It is the authors' responsibility to stop the recursive macro calls using extreme conditionals. To avoid infinite recursive macro calls, embedded JavaScript must be used directly or indirectly to provide conditionals.

IMP is designed to allow macro calls in macro calls to allow macro calls as parameters. This raises the problem of when to evaluate the actual parameters. Let us look at an example:

```
$(
    $( x = 1; $)

    $m = $(
        $(x;$)
        $argv(0)
        $(x;$)
    $)

    $m( $(++x; $) )

$)
```

The above IMP code would output '1222' if IMP took the same approach as most programming languages such as PASCAL, JAVA, and C by evaluating the parameter expression '++x' first and storing the result onto the stack before expanding macro $m. An alternative is to evaluate the expression '++x' only when expanding '$argv(0)'. In this way, the above IMP code would result in '1122'.

IMP's front-end takes the second way. One of the benefits is that IMP can simulate a 'for' loop found in most programming languages. Please see Chapter 3.4.3 for the 'for' macro example.

IMP's back-end supports both parameter evaluating methods. Please see Chapter 3.2.5.2 for IMP's back-end support for parameter evaluating.

3.2.3.8 Block

Blocks, the basic structures in the IMP source codes, are used to bind IMP elements.

The grammar for blocks is:

```
block    :=   '${' stmt* '$}'
```

3.2.3.9 Naming Scope

Since IMP allows macro definition within macro definition, it raises the problem of naming scope. IMP follows PASCAL's approach. When searching for a macro name to expand, IMP searches the macro definition within the current block. If it does not find its definition, it then searches the parent block, and so on, until it finds the macro definition or fails with any errors being reported. Let us look at an example:

```
$m1 = ${
    $m2 = ${
        $[x$]
    $}
    $m2()
$}

$m2 = ${
    $[y$]
$}

$m1()
$m2()
```

In the above example, the first macro call to $m2() will match the macro definition of $m2 inside the macro definition of $m1, the second macro call to $m2() will match the macro definition of $m2 outside the macro definition of $m1. The output of the above IMP code is 'xy'.

IMP also provides direct macro name addressing to search for macro definitions. In the above IMP code, $root.m1.m2 can be used to directly refer to the macro definition of $m2 inside the macro definition of $m1.

3.2.3.10 Macro Invocation

Some powerful macro processors, such as TRAC, are made in an active model where after the first round of macro expansion, one can perform subsequent rounds of macro expan-

sions, which take the output of the previous round of macro expansion as input, until one cannot make any change from its input to its output. IMP's front-end does not follow this approach, but rather simply performs only one round of macro expansion and ceases. The immediate benefit of this one-round neutral invocation approach is the simplicity for the macro users avoiding surprising macro expansions that are not intended while also avoiding any possible performance penalties introduced by the active invocation. A detailed discussion on active invocation and neutral invocation is covered in Chapter 3.2.4.1 with examples.

3.2.3.11 Comment

IMP uses '$<' and '$>' to enclose comments.

Here is an example of an IMP comment:

```
$< comments $>
```

The '$' style consists with those of other IMP elements.

3.2.3.12 File Inclusion

IMP provides a '$include' facility to insert another file into the current line. The inserted file content will be treated as an IMP source. This facility is admirable since it can encapsulate sets of macro definitions into separate files to serve as pre-defined macro library packages.

Here is an example of IMP file inclusion:

```
\$include fileName
```

It may be a good idea to associate '$include' with conditional and loop controls to allow complex file inclusions. However, this is not a concern in the initial version of IMP.

3.2.3.13 Summary

The following gives a complete grammar of the IMP front-end.

```
start     :=   block

block     :=   '${' stmt* '$}'

stmt      :=   argc | argv | ilscript | text | ref | smacro
               | macro | block

argc      :=   '$argc' '(' ')'

argv      :=   '$argv' '(' NUMBER ')'

ilscript  :=   '$(' JSCODE '$)'

text      :=   '$[' TEXT '$]'

ref       :=   ID '(' ( | param (',' param)* ) ')'

param     :=   argc | argv | ilscript | text | ref | block

smacro    :=   ID '=' ilscript

macro     :=   ID '=' block

NUMBER    :=   [0-9]+

ID        :=   $[a-zA-Z_]([a-zA-Z_]|[0-9])*

JSCODE    :=   JavaScript code (see note)

TEXT      :=   normal text (see note)
```

Note: Any $$ will be parsed to a single character $ as part of the JavaScript code or the normal text.

3.2.4 Powerful Back-End

The primary concern of back-end is to be as powerful as possible. The back-end takes its input, an XML representation, performs the macro expansion, and outputs the final result. Figure 3.2 gives an overview of the IMP's back-end.

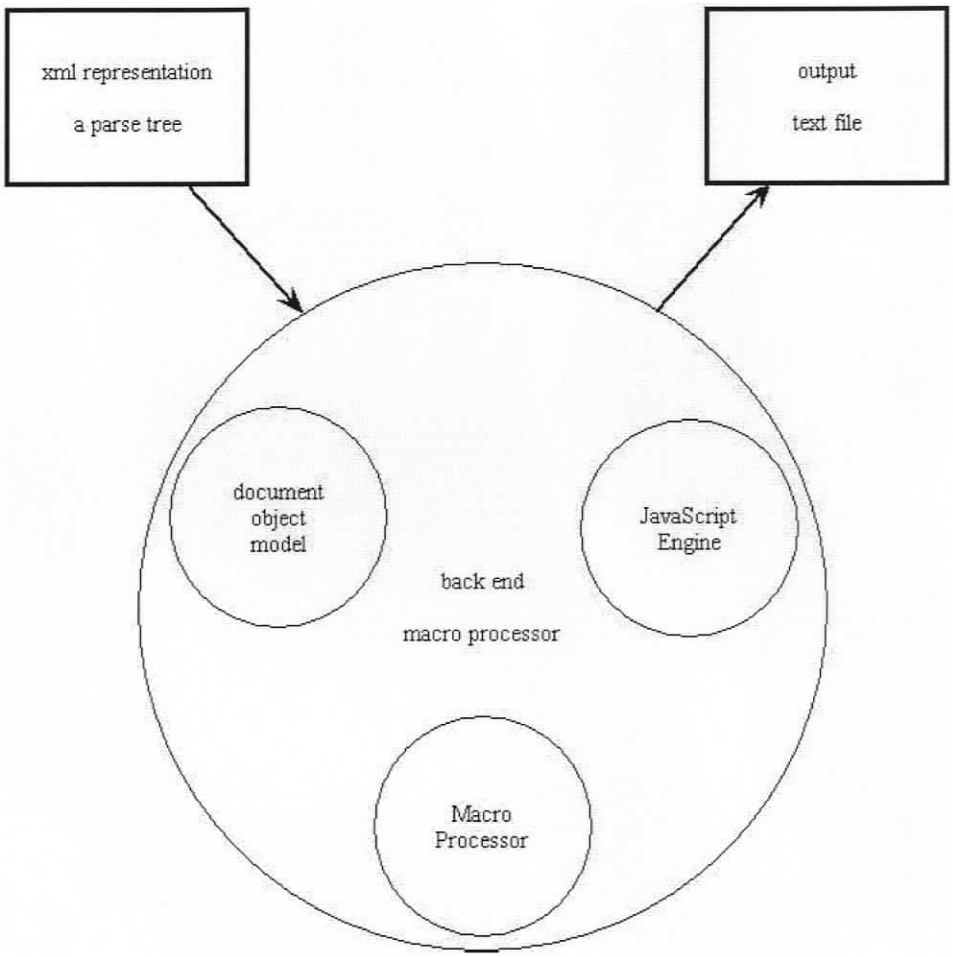


Figure 3.2. *IMP Back-End Components*

There are three primary components in the IMP's back-end, a macro processor, a JavaScript engine, and a DOM engine.

3.2.4.1 Macro Processor Component

The macro processor component of the IMP's back-end is designed to be a string oriented general-purpose macro processor reading the macro definitions and macro calls from an XML data structure, expanding the macro calls according to the macro definitions, and outputing the expanded strings. There are several design issues concerning the macro processor component. In particular, there is the passes of macro expansions where some macro processors such as TRAC and GNU's M4 allow active macro invocation where the result of the macro call is re-scanned to see if it contains further macro calls. For example, noting the following two macro definitions and two macro calls written in M4 grammar,

```
define(`x', `substr(ab')
define(`y', `cde, 3, 2)')
x``'y
```

The first line defines x to be the string 'substr(ab', and the second line defines y to be the string 'cde, 3, 2)'. The third line is a macro call x followed by another macro call y. After the first round of macro expansion, x``'y becomes 'substr(abcde, 3, 2)'. Active macro invocation macro processors like M4 will not stop here by sending the string 'substr(abcde, 3, 2)' to the output. Instead, it will search 'substr' in its macro definition list. Since 'substr' is a built-in macro definition in M4, 'substr(abcde, 3, 2)' will be further expanded which extracts a substring of 'abcde' beginning from the 3rd character with a length of 2, making 'de' the final result sent to the output.

Some other macro processors, such as GNU's Groff, do not allow active macro invocation, making every macro invocation a neutral invocation where the result is not re-scanned but rather is sent to the output. With a neutral invocation macro processor, the above example will send 'substr(abcde, 3, 2)' to the output.

Ideally, it is better for the IMP's back-end macro processor to support both active invocation and neutral invocation and allow the IMP's front-end to determine whether it is active macro invocation or neutral macro invocation. The initial version of IMP's back-end macro processor only supports neutral macro invocation, but it would not be difficult to support both neutral invocation and active invocation in a future version. IMP's front-end only supports neutral invocation, this has been discussed in Chapter 3.2.3.10.

Another issue concerning the macro processor component is the parameter evaluation time in macro calls. There are two ways to evaluate the actual arguments and assign the actual arguments to the formal parameters. One is to evaluate the actual arguments at the time of macro calls while the other is to evaluate the actual arguments at the time whenever a parameter reference is met. IMP's front-end only support the second approach which has been discussed in Chapter 3.2.3.7. To further illustrate this problem, look at another example:

```
$( var op = '+'; ""; $)
$changeOp =(
    op = bargv(0);
    op;
)
$compute =(
    var a = bargv(0);
    var b = bargv(1);
    var exp = a+op+b;
    eval(exp);
)
$print = {
    $argv(0)
    $changeOp( $argv(1))
```

```
        $argv(2)
        $[=$]
        $argv(3)
    }
    $print( $[2$], $[*$], $[3$], $compute($[2$], $[3$]) )
```

In the above example, the first line is an inline JavaScript which sets the global variable 'op' to a default value '+', while '$changeOp' changes the value of 'op'. '$compute' takes two arguments and makes a computation according to 'op'. '$print' takes four arguments, the first operant, the operator, the second operant and the result. A macro call to '$print' is requested with arguments of '2', '*', '3' and '$compute($[2$], $[3$])'.

If the evaluation of the actual arguments is performed at the time of macro calls, '$compute($[2$], $[3$])' will be evaluated first, then the result is assigned to the formal parameter $argv(3) of the macro '$print', where the output is '2*3=5'. This sometime leads to unpredictable results as shown in this example, but in actuality this is the approach that many programming languages such as PASCAL, JAVA and C take.

The other method is to evaluate the actual arguments at the time whenever a parameter reference is met. When expanding '$print', '$compute($[2$], $[3$])' will be held until '$argv(3)' is met, which occurs after the macro expansion of '$changeOp'. In this way, the operator will be changed to '+' before '$compute' is executed. The output string is '2+3=5'.

IMP's back-end supports both kinds of evaluation by specifying an 'epf' attribute in the macro calls. Please see Chapter 3.2.5.2 for details.

When the evaluation of the actual arguments is performed at the time of macro calls, the evaluation order for the actual parameters is another concern. The evaluation of the actual parameters can be from left to right (Pascal style) or from right to left (C style).

The initial version of IMP's back-end only supports left to right evaluation of the actual

parameters. Ideally, it is better to support both kinds of evaluation orders by specifying an 'order' attribute in the macro calls.

3.2.4.2 JavaScript Engine

Most existing macro processors provide some programming facilities to allow the macro users to write complex macros. For example, GNU's Groff has '.if', '.ie' (if else), '.el' (else) macros for conditional executions and '.while' for the loop control structure. GNU's M4 users can use recursions, conditionals, and a built-in macro, 'shift', for iterating through the actual arguments to a macro, to simulate a loop control.

The problem of existing macro processors is that the programming facilities they provide are more like arcane dialects. Thus their syntaxes are different from those of popular programming languages and are usually with limitations. For example, Groff uses number registers to serve as global variables. The value of these variables can only be in numbers. In M4, it is very difficult and error prone to attempt to implement nested loops although it is possible. In addition, most existing macro processors do not provide local variables and only provide very limited arithmetic operators and built-in functions.

There is a great benefit to incorporating a script engine that provides rich programming facilities and is easy to learn. JavaScript engine becomes the immediate choice. JavaScript has emerged as the scripting language of today while having gained a certain familiarity, due to its ease of learning, speed in developing, and provision of rich capabilities. JavaScript relies on ECMA(European Computer Manufacturers Association) Open Standard.

IMP's JavaScript engine evaluates every inline JavaScript code and script macro met at macro expansion time, replacing the inline JavaScript code or the script macro with the evaluated value of the last statement in JavaScript code. IMP's JavaScript engine only maintains a single runtime context for all the inline JavaScript codes and script macros. To illustrate the idea of a single runtime context, let us look at the following example:

```
$(
    x = 1;
    x;
$)

$inc = $(
    x++;
    x;
$)
$inc()
```

In the above example, both the variable 'x' in the inline JavaScript and the variable 'x' in the JavaScript macro refer to the same entity. This is because IMP only maintains a single runtime context. The last statement of inline JavaScript or JavaScript macro determines its output. The inline JavaScript will be expanded to string '1', and the macro '$inc()' will be expanded to string '2'. Thus the final output will be '12'.

Technically, it is possible to make a design that has multiple runtime contexts, but that would not be as useful as having a single runtime context.

3.2.4.3 DOM

Another benefit to IMP that comes with JavaScript is a DOM is provided to allow macro authors to manipulate the macro processing in JavaScript.

Figure 3.3 summarizes the IMP Object Hierarchy.

IMP's Document Object Model is relatively simple compared with the DOM in web browsers and the DOM specified by W3C [33]. The uppermost level in IMP's hierarchy is the 'doc' object, an abstract object that refers to a document. There are two child objects in doc's properties, 'i' (input) and 'o' (output). 'root' is the base object of the input to IMP having the uppermost level macro definitions as its children. Each macro definition has its own local macro definitions as its children. By 'doc.i.root.' followed by the names of macro definitions, one can refer to each individual macro definition in JavaScript. For example:

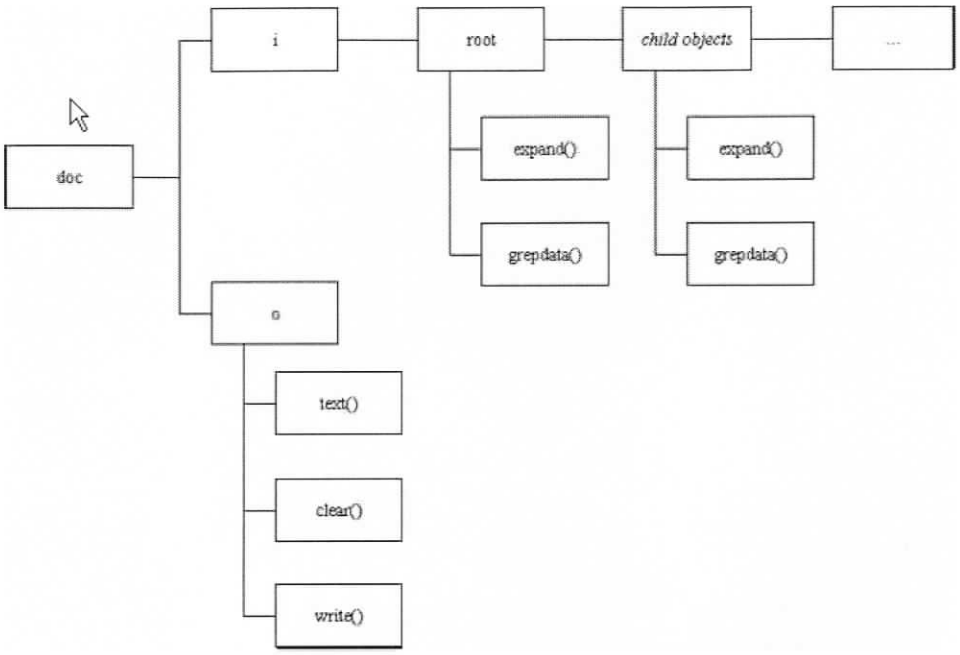


Figure 3.3. *IMP Object Hierarchy*

```
\${
    \$m1 = \$\{
        \$m2 = \$\{
            \$[The key is here.\$]
        \$\}
    \$\}
    \$(
        var r="";
        r = doc.i.root.m1.m2.expand();
        r;
    \$)
\$\}
```

'doc.i.root.m1.m2' refers to the macro definition $m2.

Each object under 'doc.i', including object 'root', has two methods, 'expand()' and 'grepData()'. 'expand()' is used to expand the macro, while 'grepData()' is used to return the macro definition in text without expansion.

In IMP's initial version, 'doc.i' is mainly used as a read-only object. In future versions

of the IMP, when active invocation is taken into consideration, 'doc.i' will be designed as a read/write object to allow dynamically changing macro definitions.

'doc.o' refers to the output after macro expansion. Method 'clear()' clears the output buffer of IMP, while method 'text()' returns the text of the current output buffer of IMP, and finally, method 'write()' appends texts to the current output buffer of IMP.

3.2.5 An XML Interface

An XML style structure is chosen as the interface between the front-end and the back-end of IMP. The structure could actually be in any proprietary form. An XML style structure is chosen because XML is based on a hierarchical structure that can express the structure of IMP source codes and because XML will likely remain the most popular in the near future. With a plethora of XML tools, analyzing, manipulating, and generating XML style structures become easy, routine tasks. The XML interface of IMP is expressed in a DTD structure which contains rules on macro definitions, macro references, parameter definitions and parameter references.

3.2.5.1 Macro Definition

There are four kinds of block level macro definitions, 'block', 'ilblock', 'script' and 'ilscript'. The difference between inline and non-inline macro definitions is that inline is expanded immediately whenever processing occurs while having no name, whereas non-inline is expanded only by explicit reference and usually has a name.

```
<!-- block macro definition -->

<!ELEMENT block      (#PCDATA | argc | argv | ref | ilscript
                     | script | ilblock | block)*>

<!ATTLIST block      id  CDATA \#REQUIRED>
```

'block' defines a block which contains normal text (#PCDATA), the number of arguments (argc), argument reference (argv), macro calls (ref), or any other blocks (ilscript |

script | ilblock | block). 'block' has an 'id' attribute which is the name of the block to be referred.

```
<!-- inline block macro definition -->

<!ELEMENT ilblock    (#PCDATA | argc | argv | ref | ilscript
                     | script | ilblock | block)*>

<!ATTLIST ilblock    id CDATA #REQUIRED>
```

'ilblock' defines an inline block which is almost the same as 'block' except that it is used as special block by assigning special value to its 'id' attribute. For example, the outermost block will be translated into an 'ilblock' with 'root' as its 'id' value.

```
<!-- JavaScript macro definition -->

<!ELEMENT script     (#PCDATA)>

<!ATTLIST script     id CDATA #REQUIRED>

<!ATTLIST script     fn CDATA #REQUIRED>

<!ATTLIST script     ln CDATA #REQUIRED>
```

'script' defines a JavaScript block which contains JavaScript codes. 'script' has an 'id' attribute which is the name of the block to be referred and which has the same functionality as the 'id' attribute of 'block'. The 'fn' and the 'ln' attribute indicate the file name and the beginning line number of this script. These attributes are particularly useful for reporting the exact position when errors occur in the JavaScript codes.

```
<!-- inline JavaScript macro definition -->

<!ELEMENT ilscript   (#PCDATA)>

<!ATTLIST ilscript   id CDATA #REQUIRED>

<!ATTLIST ilscript   fn CDATA #REQUIRED>

<!ATTLIST ilscript   ln CDATA #REQUIRED>
```

'ilscript' defines an inline JavaScript block, having the same attributes as 'script'. Since here the 'ilscript' has also an 'id', the back-end treats 'ilscript' and 'script' in the same manner. The front-end gives the inline JavaScript a special generated name.

3.2.5.2 Macro Reference

```
<!-- reference to macros -->
<!ELEMENT ref        (param)*>
<!ATTLIST ref        id CDATA #REQUIRED>
<!ATTLIST ref        epf CDATA #REQUIRED>
```

'macro reference' is another name of 'macro call'. A macro reference has zero or more parameters which is described in Chapter 3.2.5.3. The 'id' attribute of macro reference specifies the name of the macro to be expanded. 'id' should have the same CDATA value as its macro definition. The 'epf' attribute of macro reference specifies the time of actual argument evaluation in macro calls. Its value is either 'true' to evaluate the actual arguments at the time of macro calls or 'false' to evaluate the actual arguments at the time whenever a parameter reference is met. The details of the time of actual argument evaluation in macro calls are described in Chapter 3.2.3.7 and Chapter 3.2.4.1.

3.2.5.3 Parameter Definition

```
<!-- parameter definition -->
<!ELEMENT param      (#PCDATA | argc | argv | ref | ilscript
                     | script | ilblock | block)*>
```

'param' specifies the arguments passed with macro expansion, and can be normal text (#PCDATA), the number of arguments(argc), argument reference(argv), macro calls(ref), or blocks(ilscript | script | ilblock | block).

3.2.5.4 Parameter Reference

```
<!-- reference to the number of arguments -->
<!ELEMENT argc      EMPTY>
<!-- reference to arguments -->
<!ELEMENT argv      EMPTY>
<!ATTLIST argv      i CDATA #REQUIRED>
```

Parameter references are used to refer to the parameters passed with macro expansion. 'argc' refers to the number of arguments, while 'argv' refers to the argument indexed by attribute 'i' which starts from zero.

3.2.5.5 Summary

The Appendix A.1 presents a complete XML interface of IMP in a DTD structure.

3.3 Implementation

This section covers some important issues in IMP implementation.

Currently, IMP is implemented only on Linux. It is possible to port IMP to other platforms. IMP is written in C++ and is compiled under GNU's GCC. The source codes are distributed in several sub-directories, three of which are front-end, backend, and shell. Other directories are related to documentation, examples, tests and other miscellaneous items.

3.3.1 Front-End

Writing a front-end is relatively simple. Lex and YACC, used to parse and analyze the input, are tools that make the front-end even simpler to implement.

Comments and file inclusions are processed at the lexical analysis phase. A class 'idList' manages all the macro IDs in a linked list. A standard template 'stack' of the STL (Standard Template Library) is used to track on nested file inclusion. Half of the front-end codes are related to syntax error handling. A simple main.cpp is provided in order that the frond-end part can be compiled and linked to an executable for testing without disturbing the back-end and the shell. This main.cpp is not used when the front-end is linked with the back-end and the shell to make the IMP executable.

3.3.2 Back-End

Writing a back-end is an involved undertaking. To ease the implementation, several libraries are used such as libgdome, also known as gdome2, which is a DOM level 2 C implementation, and SpiderMonkey, Mozilla's C implementation of JavaScript.

The back-end codes are implemented in several classes. Class 'macroProcessor' encapsulates the macro processor component, while Class 'jsengine' encapsulates the interface between macro processor component and JavaScript engine. Class 'rtContext' encapsulates the stack frame for the macro expansion runtime, while Class 'xmldom' encapsulates the operations with XML structures, and finally Class 'erroreport' encapsulates the error handling.

The macro processor is designed to faciliate reentrance. Method 'macroProcessor::evalBlock' is a recursive function that evaluate blocks. Class 'jsengine' internally maintains only one copy of runtime context. IMP's DOM engine objects and their methods are also implemented in this class. Method 'jsengine::eval' evaluates inline JavaScript codes and Script macros under a single runtime context. Class 'erroreport' processes any runtime errors with macro expansion. The runtime errors associated with JavaScript are handled in class 'jsengine', while the syntax errors are handled in the front-end. Since the macro processor engine is designed to be capable of reentrance, class 'rtContext' is used to hold the parameters passed into the blocks in a stack frame style.

A simple main.cpp is also included in order that the back-end part can be compiled

and linked to an executable for testing without disturbing the front-end and the shell. This main.cpp is not used when the back-end is linked with the front-end and the shell to make the IMP executable.

3.3.3 Shell

The source codes in sub-directory 'shell' provide the 'main' function of IMP and parse any command line options, while connecting the front-end and the back-end together.

3.4 Examples

This section includes several examples to demonstrate some features of the IMP and is intended to give the reader a familiarity of the look and feel of the IMP programming.

3.4.1 Example - Script Macro

This example illustrates accessing parameters passed in a script macro.

```
$<
*******************************************************
    Example : Script Macro
*******************************************************
$>
${
    $m1 = $(
        var s = "";
        s += "argc = "+bargc()+"\n";
        for (var i=0; i< bargc(); i++) {
            s += "argv["+i+"] = ";
            s += bargv(i);
            s += "\n";
        }
        s;
    $)

    $< call m1 with some parameters $>
    $m1(
        $[parameter 1$],
        $[parameter 2$],
```

```
        $[parameter 3$],
        $[parameter 4$],
        $[parameter 5$],
        $[parameter 6$]
    )
$}
```

The output is:

```
argc = 6
argv[0] = parameter 1
argv[1] = parameter 2
argv[2] = parameter 3
argv[3] = parameter 4
argv[4] = parameter 5
argv[5] = parameter 6
```

3.4.2 Example - If

This example illustrates a method to simulate the 'if-else' construct usually provided by programming languages.

```
$(
    $<
    ******************************************************
        Example : if
        Usage   : $if(condition, body1, body2)
    ******************************************************
    $>
    $if = $(
        var r = "";
        if (bargv(0) == "true") {
            r = bargv(1);
        }
        else {
            r = bargv(2);
        }
        r;
    $)

    $< test 'if' $>
    $if(
        $( 22*24 < 23*23 $),
        $[ 22*24 is less than 23*23.$],
```

```
        $[ 22*24 is greater than 23*23.$]
    )
$}
```

The output is:

```
22*24 is less than 23*23.
```

3.4.3 Example - For

This example illustrates a method to simulate the 'for' construct usually provided by programming languages. Using a similar method, 'while' and 'repeat' can also be implemented.

```
$(
    $<
    ******************************************************
        Example : for
        Usage   : $for(init, condition, update, body)
    ******************************************************
    $>
    $for = $(
        var r = "";
        bargv(0);
        for (; bargv(1)=="true"; bargv(2)) {
            r += bargv(3);
        }
        r;
    $)

    $< test 'for' $>
    $for(
        $( var i=0 $),
        $( i < 20 $),
        $( i++ $),
        $[a$]
    )
$}
```

The output is:

```
aaaaaaaaaaaaaaaaaaaa
```

3.4.4 Example - Macro Reference

This example illustrates one method of macro reference. This method is to directly reference to the macro name by specifying its name path.

```
$<
*******************************************************
    Example : Macro Reference
*******************************************************
$>

${
    $m1 = ${
        $m2 = ${
            $[The key is here.$]
        $}
    $}

    $root.m1.m2()
$}
```

The output is:

```
The key is here.
```

3.4.5 Example - Macro Reference via DOM

This example illustrates the method of macro reference in JavaScript. This method is based on IMP's DOM.

```
$<
*******************************************************
    Example : Macro Reference In DOM
*******************************************************
$>
${
    $m1 = ${
        $m2 = ${
            $[The key is here.$]
        $}
    $}
```

```
    $(
        var r="";

        r = doc.i.root.m1.m2.expand();
        r;
    $)

$}
```

The output is:

```
The key is here.
```

3.4.6 Example - DOM

This example illustrates some usages of IMP's DOM.

```
$<
*******************************************************
    Example : DOM
*******************************************************
$>
${
    $[The key is here.$]

    $(
        var r=doc.o.text(); // copy output buffer into var r

        doc.o.clear();      // clear output buffer

        doc.o.write("The output is : ");

        doc.o.write(r);

        "";                 // return empty
    $)
$}
```

The output is:

```
The output is : The key is here.
```

3.5 Applications

The ISE syntax is quite tedious to learn and understand, thus it is a great advantage to provide macro packages which automatically generate ISE codes and which allow intensional authors to bypass the arcane ISE syntax. To design a fully functional macro package that is equivalent to ISE may be too ambitious, but a lightweight wrapper for frequently-used ISE constructs may be sufficient enough in practice. IMP is designed expressively for this purpose. This section describes some ISE generating macro packages written in IMP.

3.5.1 Enhanced IML

The current approach to write intensional pages by IML is limited by the poor macro facilities of Groff. The rich facilities of IMP enhance ease of use and lend a versatility for authoring. IMP makes it easier to separate the complicated multidimensional web page authoring problem into two distinct parts, presentation and content. The author focuses on the content part with the detail of presentation part being hidden by the IMP macros. With the powerful macro facilities of IMP, incorporating more sophisticated presentation tools such as Cascading Style Sheets (CSS) into HTML is now feasible.

Figure 3.4, 3.5, 3.6 and 3.7 show examples of three re-implementations of 'drop text' by IMP macros with CSS.

Figure 3.4 has several improvements over the implementation in IML, although the look is quite similar. One improvement is that the drop texts can be nested in any levels with structures easy to write which is depicted below:

```
......

$dt1(
    $[Lunch$],
    ${
        $dt1(
            $[GRILLED FRESH ATLANTIC SALMON$],
            $[Sweet Corn and Fennel Melange,  ... $]
        )
        $dt1(
```

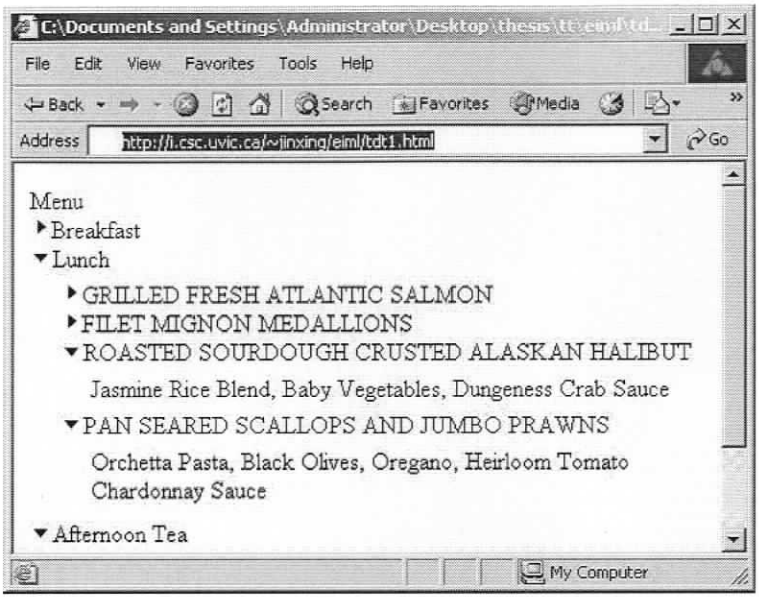


Figure 3.4. *Drop Text (a)*

```
            $[FILET MIGNON MEDALLIONS$],
            $[Asparagus, Roasted Shallots, ... $]
        )
        $dt1(
            $[ROASTED SOURDOUGH CRUSTED ALASKAN HALIBUT$],
            $[Jasmine Rice Blend, ... $]
        )
        $dt1(
            $[PAN SEARED SCALLOPS AND JUMBO PRAWNS$],
            $[Orchetta Pasta, Black Olives, ... $]
        )
    $}
)

......
```

Another improvement is that the expanded text can now be delivered to a browser only when requested which is extremely important for performance. This is implemented by taking the idea of Remote Scripting, by which a browser and a server-side application can exchange data without reloading the whole page. Remote Scripting offers the beneficial functionality in the use of 'IFRAME' tag in a web page serving as an internal buffer on the

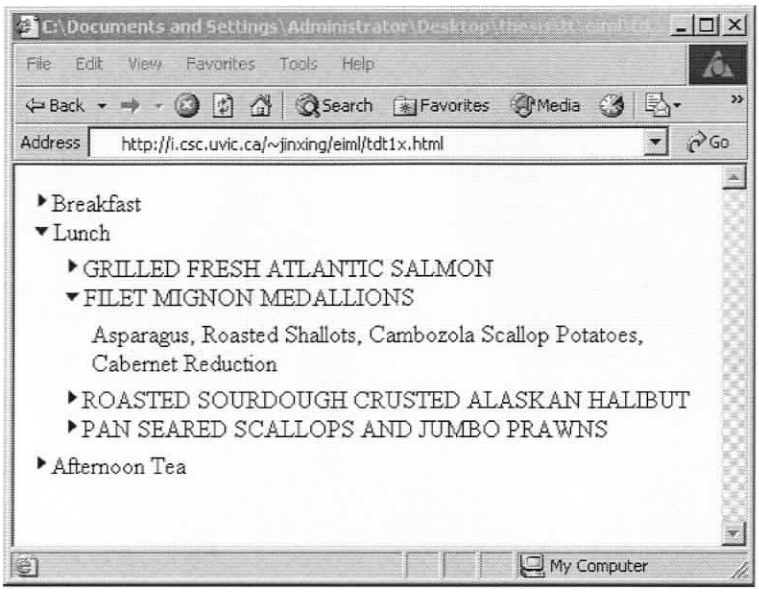


Figure 3.5. *Drop Text (b)*

client side. The following code defines an 'IFRAME' as an internal buffer:

```
$[<iframe id="impGetSeverData" style="width:0px;
    height:0px; border: 0px;" src=""></iframe>$]
```

When a drop text is requested, the 'src' property of the 'iframe' is assigned to the appropriate URL by client side scripts. Usually with version information as the suffix, this URL points to an ISE script generated by IMP macros.

There are two benefits of this new implementation. One is the provision of a more appealing visual effect. Since the page now does not need to be reloaded into browsers when clicking on the expand or shrink icons, there is no flicker side effect to the human eyes while the current position in the browser remains. Another benefit is the improved network bandwidth utilizing. In writing a book of 1M bytes size in drop text style, for example, now the browser would no longer need to load the whole 1M book at the beginning.

Figure 3.5 is almost the same as in Figure 3.4 except that only one item can be expanded at the same level in figure 3.5 whereas multiple items can be expanded at the same level in

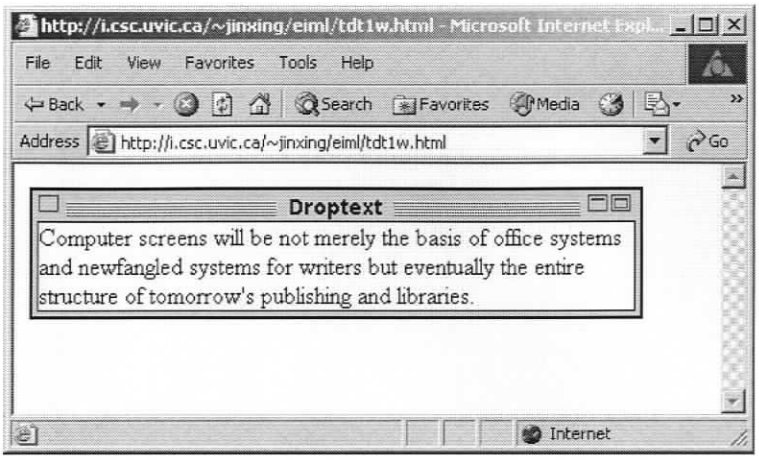


Figure 3.6. *Drop Text (c)*

figure 3.4.

Figure 3.6 and figure 3.7 show a drop text in a window. Clicking on the minimize icon on figure 3.6 leads to figure 3.7, and clicking on the maximize icon on figure 3.7 leads to figure 3.6. Clicking on the close icon at the left top of a window dismisses the window.

Figure 3.8, figure 3.9, and figure 3.10 show a re-implementation of stretch text. Clicking on the expand icon on figure 3.8 leads to figure 3.9. The shading lasts for a few seconds then changes to normal. Clicking on the expand icon in figure 3.9 leads to figure 3.10, and clicking on the shrink icon in figure 3.9 returns one to figure 3.8.

In addition to drop text and stretch text, there are many other visual elements in IML such as forms, menus, slides, pop-up windows, strolls, to name a few. With the powerful facilities of IMP, enhancing or adding more of these elements is now easier. Perhaps the only limitation is one's imagination.

There are many ways to enhance IML for intensional web page authoring. Incorporating CSS is only one of them. Unfortunately, this approach is limited by the incompatibilities of current main browsers. The examples shown here were tested on Microsoft Internet Explorer 5.5. More effort is required to allow a functionality on multiple browser platforms.

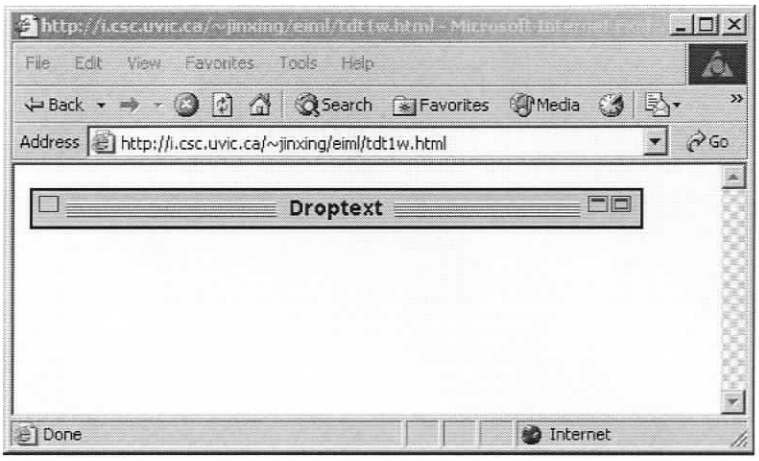


Figure 3.7. *Drop Text (d)*

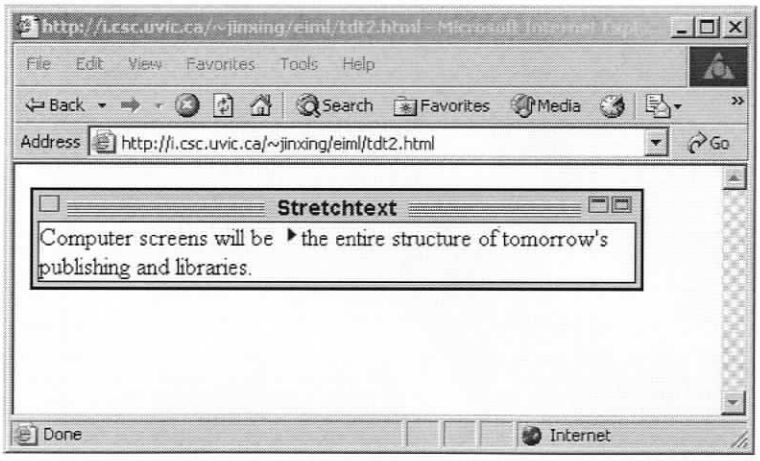


Figure 3.8. *Stretch Text (a)*

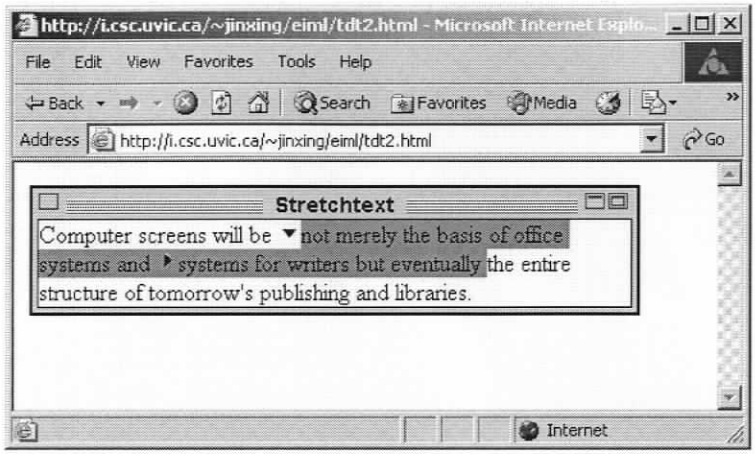


Figure 3.9. *Stretch Text (b)*

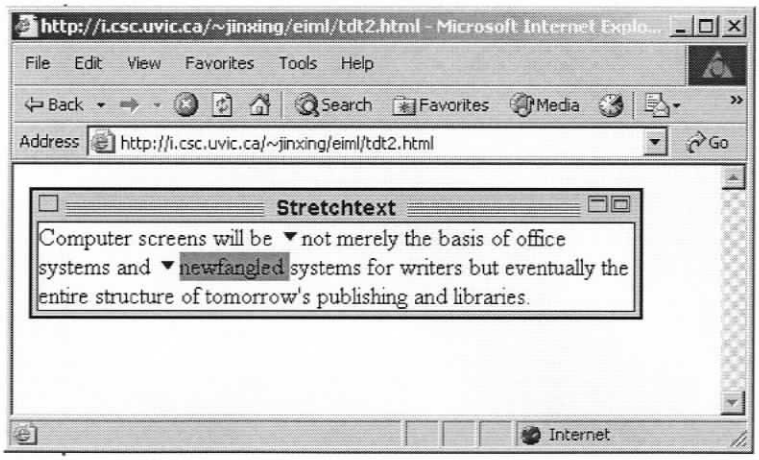


Figure 3.10. *Stretch Text (c)*

3.5.2 WIMPAS

WIMPAS is based on IMP.

Traditionally, intensional markup authoring is performed on Linux. In a terminal window, the author logs on to the Linux system, types an IMP document in his favorite editors (Vi, Nedit or any software that provides an editor), compiles the IMP document to ISE, and finally launches a browser to view the ISE generated. The author is required to have a decent knowledge of the UNIX system. In addition, switching between the shell, the editor, and the browser is tedious and time-consuming.

With WIMPAS, intensional markup authoring becomes more efficient and more pleasurable. WIMPAS provides a web-based GUI editor with a toolbox that allows entering the pre-defined macros and IMP elements by mouse clicks. This even lessens the IMP learning curve because the author now does not have to remember anything about the IMP syntax and the pre-defined macro names. The toolbar comes into use here, where one simply clicks on it to save time in typing.

Less typing is only one of the benefits that WIMPAS offers. Now, collected IMP reference, ISE reference, core JavaScript guide and reference, simple project management, IMP and ISE editing, ISE rendering, area all integrated into one web page. With the help of WIMPAS, intensional markup authors now only need to launch a web browser to write IMP scripts and view the generated ISE page instantly.

With more pre-defined IMP macros being added into WIMPAS, it is expected that intensional markup authors will no longer need to know the details of ISE. Eventually, writing intensional markup could be just sequences of mouse clicks and the requirement of a series of forms filled out on the web.

Figure 1.1 is a screen snapshot of WIMPAS. References and the toolbox are on the left side while the top right side lists the steps for usual IMP authoring: 'select a project', 'view pre-defined macros', 'edit user-defined macros', 'edit imp', 'generate ise' and 'view ise'.

WIMPAS consists of a project manager, an editor, a portal and a multitude of references. While some of them were written in Perl, many were developed by using IMP macros.

WIMPAS can be reached at http://i.csc.uvic.ca/~jinxing/wimpas/index.html.

3.5.3 Intensional Spreadsheet

In [34], W. Du and W. W. Wadge described a 3D spreadsheet based on intensional logic. One key feature of this spreadsheet design is that the whole spreadsheet is considered a single entity with three dimensions, that of two spatial dimensions and one temporal dimension. By extending the idea of the 3D spreadsheet, an intensional spreadsheet has been designed with ISE and IMP.

Dimension definitions define dimension names and their possible values. Dimensions are not limited to spatial and temporal dimensions. Any number of dimensions can be defined. Dimension definition is accomplished by an IMP macro $dim, shown in the following examples:

```
$dim(   $[sb$],     $vanilla(),             $[fille$],
                                            $[garcon$],
                                            $[professeur$],
                                            $[etudiant$])
$dim(   $[sb$],     $[sbct:pro+sbn:plu$],   $[nous$],
                                            $[vous$],
                                            $[ils$],
                                            $[elles$])
$dim(   $[sb$],     $[sbct:pro+sbn:sin$],   $[je$],
                                            $[tu$],
                                            $[il$],
                                            $[elle$])
```

In the above IMP example, a dimension 'sb' is defined in three different contexts. 'fille', 'garcon', 'profeseur' and 'etudiant' are possible values in vanilla context. 'nous', 'vous', 'ils' and 'elles' are possible values in context 'sbct:pro+sbn:plu'. 'je', 'tu', 'il' and 'elle' are possible values in context 'sbct:pro+sbn:sin'.

Cell definitions define the intensional cell content. That is, the cell content depends on the context. The content can be texts, formulas, or dimensions, as shown in the following examples:

```
$cell_text($[1$], $[1$],    $vanilla(),     $[Sujet$])

$cell_dim ($[1$], $[4$],                    $[sb$])

$cell_text($[12$], $[1$],   $vanilla(),     $[=m1()$])

$cell_text($[12$], $[2$],   $vanilla(),     $[$],
                            $[vbf:neg$],    $[ne@ $])

$cell_text($[16$], $[2$],   $vanilla(),     ${
    $[=$] $[filter3($] $cell($[15$],$[2$]) $[)$] $})
```

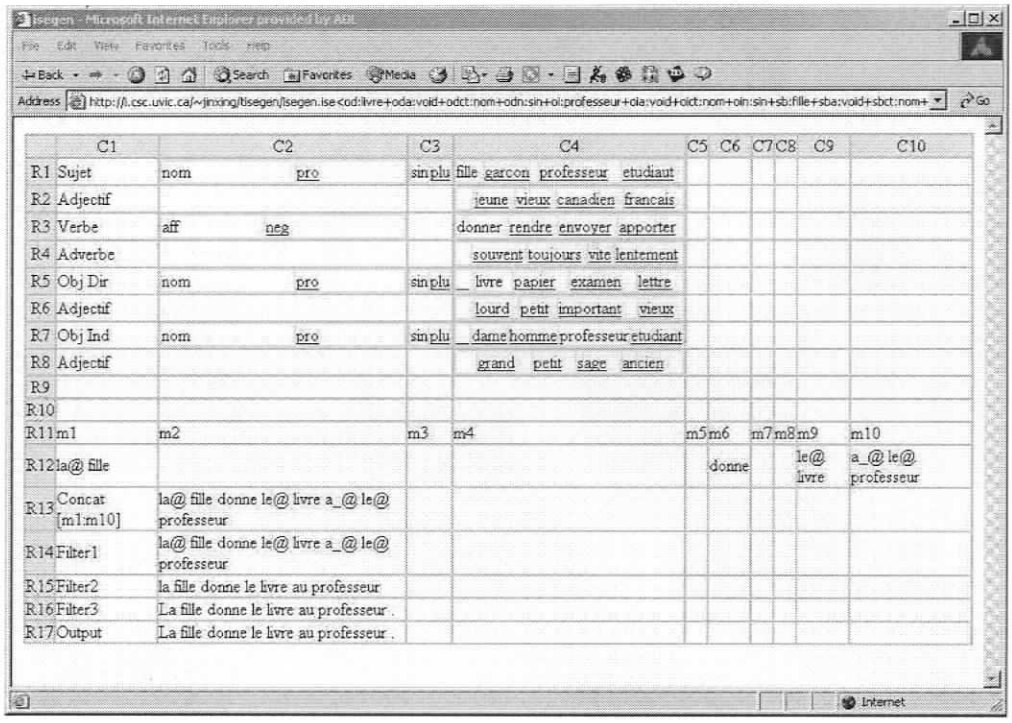
In the above IMP example, cell[1,1] is defined as text 'Subjet' under vanilla context. Cell[1,4] is defined as dimension 'sb'. Cell[12,1] is defined as a formula '=m1()'. A formula always starts with a '='. 'm1()' is a built-in ISE function. The content of this cell is computed by evaluating the formula. Cell[12,2] is defined to be an empty text under vanilla version or 'ne@ ' under context 'vbf:neg'. Cell[16,2] is defined as a formula '=filter3(cell[15,2])'. Referring to other cells in a formula takes the intensional meaning of the cells referred.

A HTML form of the spreadsheet can be produced by a few IMP macros, shown as below:

```
$html_begin($[ isegen $])
$html_write($[ spreadsheet(1,1,17,10) $])
$html_end()
```

In the above example, a part of the spreadsheet starting from cell[1,1] with 17 rows and 10 columns is generated into a HTML form, with the result shown in Figure 3.11. Row 1 to Row 8 are context switches. Clicking on any URLs in these rows will result in a context switch. The content of some cells will switch to that of the new context and the context switches will also switch to appropriate URLs for further context switch requests.

The examples shown in this subsection are taken from a re-implementation of the French Sentences Makers by Bill and Christine Wadge [4]. The previous ISE implementation is performed in over 500 lines of ISE code. Both programming and the comprehension



	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
R1	Sujet	nom pro	sin plu	fille garcon professeur etudiant						
R2	Adjectif			jeune vieux canadien francais						
R3	Verbe	aff neg		donner rendre envoyer apporter						
R4	Adverbe			souvent toujours vite lentement						
R5	Obj Dir	nom pro	sin plu	livre papier examen lettre						
R6	Adjectif			lourd petit important vieux						
R7	Obj Ind	nom pro	sin plu	dame homme professeur etudiant						
R8	Adjectif			grand petit sage ancien						
R9										
R10										
R11	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10
R12	la@ fille					donne			le@ livre	a_@ le@ professeur
R13	Concat [m1:m10]	la@ fille donne le@ livre a_@ le@ professeur								
R14	Filter1	la@ fille donne le@ livre a_@ le@ professeur								
R15	Filter2	la fille donne le livre au professeur								
R16	Filter3	La fille donne le livre au professeur .								
R17	Output	La fille donne le livre au professeur .								

Figure 3.11. *Intensional Spreadsheet View 1*

of 500 lines of ISE code are extremely difficult because it combines HTML programming, ISE programming and French grammar logic in one piece of paper. The re-implementation is undertaken in a simple intensional spreadsheet. The intensional spreadsheet approach greatly releases the burden of ISE and HTML programming on authors. The authors can focus on defining French grammar rules by providing cell definitions. The intensional spreadsheet system is in charge of many of the tasks of ISE and HTML programming.

Figure 3.12 shows the spreadsheet by screening the spreadsheet in four views.

The whole spreadsheet for French Sentence Maker consists of 40 dimension definitions and approximately 15 key cell definitions.

The definitions in the intensional spreadsheet are translated into complex ISE scripts. The following shows some of the generated ISE code by IMP:

```
......

function sb<>() {
    local(@r);
    @r = {
        "fille", "garcon", "professeur", "etudiaut"
    };
    return @r;
}

function sb<sbct:pro+sbn:plu>() {
    local(@r);
    @r = {
        "nous", "vous", "ils", "elles"
    };
    return @r;
}

function sb<sbct:pro+sbn:sin>() {
    local(@r);
    @r = {
        "je", "tu", "il", "elle"
    };
    return @r;
}

......

function cell<row:r1+col:c4>() {
    local($r);
```

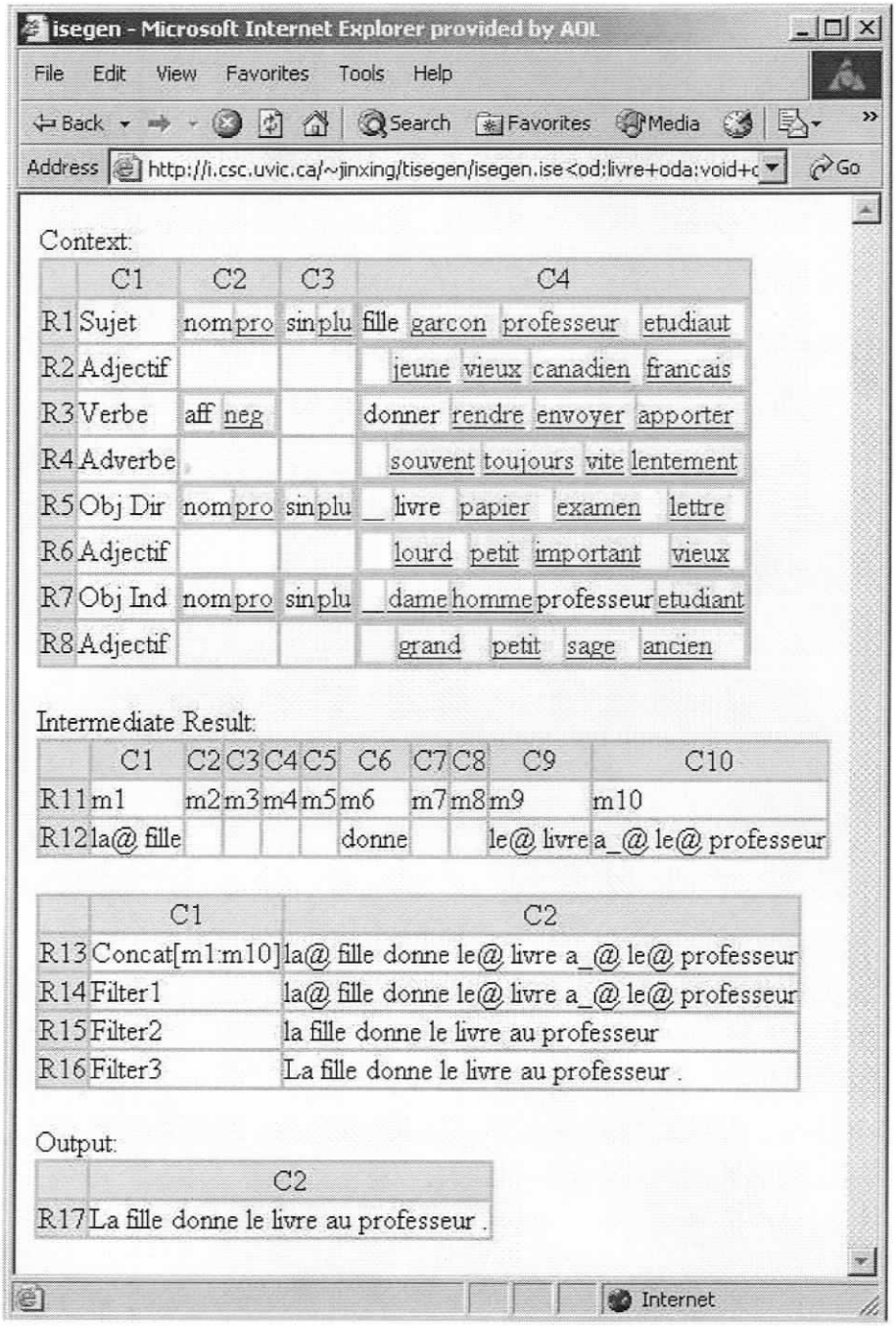


Figure 3.12. *Intensional Spreadsheet View 2*

```
    $r = sb_html();
    return $r;
}

function cell<row:r12+col:c2>() {
    local($r);
    vswitch {
        <> {$r = "";}
        <vbf:neg> {$r = "ne@ ";}
    };
    return $r;
}

function cell<row:r16+col:c2>() {
    local($r);
    vswitch {
        <> {$r = filter3(cell<row:r15+col:c2>());}
    };
    return $r;
}

......
```

An interactive intensional spreadsheet system based on web can be further enhanced with a rich set of built-in functions and a more complex formula expression evaluator. Users can define dimensions and cells and view the spreadsheet interactively in a web browser in almost the same way as in Microsoft Excel.

3.5.4 Other Applications

IMP can also be used as a general macro processor. It can be used as a text manipulator, as a macro processor for assembler languages, or even as an ICC generator. Offering a simple syntax and powerful computing facilities, IMP is expected to have a comprehensive list of applications in the near future.

Chapter 4

ICC - An Intensional C Compiler

Intensional C, a C language based on ANSI C with intensional extensions, is one of the next steps in intensional markup authoring after ISE (based on Perl). Although ICC is intended to be a complement rather than a replacement of ISE, as the creation of ICC draws many lessons from P. Swoboda's work in the creation of ISE, ICC has some noticeable advantages that ISE does not offer.

Similar to Perl, one advantage of ISE is its concise syntax for expressing. At times this may lead to obscure source codes, lengthen the learning curve, and limit the application of ISE. Based on ANSI C, ICC exhibits a clarity in syntax and semantic meanings and facilitates easy learning in comparison to ISE. In term of performance, ISE programs are considered as scripts rather than executables as they require an interpretation by the ISE interpreter at runtime. The runtime speed, however, is not desirable in some situations. ICC programs, which are compiled into machine instructions by ICC compiler, usually run quicker than their ISE equivalents. In addition, ISE source codes and an ISE interpreter must be delivered in order to run the ISE programs. Occasionally this is not desired for reasons such as security and source code protection. ICC programs, which are compiled into machine instructions, can be delivered to minimize these concerns.

The intensional C language extends the ANSI C grammar with only a few modifications that mainly focus on identifiers and statements. These modifications are shown by the following grammar and regular expression.

identifier := *[a-zA-Z_][0-9a-zA-Z_]**

version ⟶ *'@' version_string '@'*

ID ⟶ *identifier [version]*

statement ⟶ *ID ':' statement*

⟶ *case constant_expression ':' statement*

⟶ *default ':' statement*

⟶ *[expression];*

⟶ *if '(' expression ')' statement*

⟶ *if '(' expression ')' statement else statement*

⟶ *switch '(' expression ')' statement*

⟶ *while '(' expression ')' statement*

⟶ *do statement while '(' expression ')' ;*

⟶ *for '(' [expression] ; [expression] ; [expression] ')' statement*

⟶ *break;*

⟶ *continue;*

⟶ *goto ID ;*

⟶ *return [expression];*

⟶ *vswitch '{' vswitch_statement '}'*

⟶ *compound_statement*

compound_statement ⟶ *'{' declaration statement '}'*

vswitch_statement ⟶ *case version ':' statement*

⟶ *default ':' statement*

Note 1: The intensional versioning scheme and the 'refine to' algorithm used in ICC are similar to those in ISE, which are discussed in Appendix B.

Note 2: This grammar is good for a LL parser such as ICC, but not for LR parsers. For LR parsers, a LR grammar can be derived from this grammar.

Identifiers now can be followed by a version string quoted in '@'. The followings give a few examples to illustrate intensional C identifiers.

```
/* Example - Intensional Variables */

int total@@ = 1;
int total@type:auto@ = 2;
int x = total;

/* Example - Intensional Functions */

int compute@op:add@(int a, int b)
{
    return a+b;
}
int compute@op:sub@(int a, int b)
{
    return a-b;
}
int main()
{
    return compute(5,3);
}
```

A default version, 'total@@', is initialized to '1', and a 'type:auto' version of 'total' is initialized to '2'. The value of 'x' is assigned to be the value of 'total' that depends on the context. A 'op:add' version and a 'op:sub' version of function 'compute' are defined. The return value of the 'main' depends on the context. If the context is 'op:add', an '8' is returned. If the context is 'op:sub', a '2' is returned.

A 'vswitch' statement is added along with a special form of 'case' statement. Keywords 'break' and 'default' are also valid inside a 'vswitch' structure and possess the same semantic meaning as their cousins in 'switch'. The following example illustrates the 'vswitch' statement.

```
/* Example - vswitch */

vswitch {
    case @language:english@ : {
        x = 1;
        break;
```

```
        }
        case @language:french@ : {
            y = 1;
            break;
        }
        default : {
            z = 1;
        }
    }
```

Depending on context, '1' is assigned to 'x' if the dimension 'language' in context is 'english', 'y' if the dimension 'language' in context is 'french', or 'z' for other languages.

Although the grammar looks particularly simple, the semantic meanings in reality are very difficult to implement, especially with regard to performance considerations. To avoid performance penalty, ICC takes a selectable intensional identifier scheme instead of *'education'*, the common computation model found in declarative intensional languages such as Lucid and ISE. Programmers can now decide whether an identifier is intensional to reduce the overload introduced by intensional versioning.

Interested readers may desire to read Appendix B for the design and implementation details of ICC, where one also locates pertinent ICC examples and applications. For a full syntax grammar for ANSI C, please refer to [35] and [36].

Chapter 5

Future Work

5.1 Enhancement to IMP

There are several means of rendering IMP more powerful and easier to use. Some possible improvements have been mentioned in Chapter 3, while more examples will follow here. Named parameters, multiple output diversions, and user customizable delimiters imbue IMP with greater convenience while an active invocation gives IMP greater macro processing capacity.

Another way of adding power to IMP is to allow macros to be inserted or modified at expansion time. A few existing macro processors have the ability to rename a macro and undefine a macro. Experience with these macro processors has shown that this sometimes leads to obscure runtime errors. Incorporating this feature into IMP would predictably enhance its error detection abilities.

ISE could also be embedded in IMP just as JavaScript, thus future IMP may indeed embed multiple script languages.

5.2 Intensional Script Engine

The intensionality of IMP could also be achieved by embedding an intensional JavaScript engine. While no such engine exists at present, an intensional JavaScript engine would offer significant meaning to intensional programming. ISE could also be implemented in

an embedding engine form. Either an intensional JavaScript engine or an ISE engine would greatly ease the work of intensional applications.

5.3 Enhancement to ICC

There are several methods to enhance both the intensional C language and the ICC compiler. A more sophistic data structure for version space such as the threaded AVL tree in ISE will greatly improve the performance of intensional version operations. A standardized intensional version operator library is critical here since more intensional markup languages will invariably be offered in the near future. In addition, a dynamic version implementation and intensional types would add power to ICC. The dynamic version is described in Chapter B.2.4.

5.4 Intensional Type

During the development of intensional C language, one interesting problem remained unanswered, that is the question of intensional types. The answer could be a form like the union structure in C language, or a form such as the omnipotent 'var' data type in JavaScript or a form such as the object in OO (Object-Oriented) languages.

5.5 Intensional OO Language

The idea of intensional OO languages is not a new idea. Bill Wadge compares intensional computation and object oriented computation in [37]. In [13], objects in OO computing is analogized to filters in Lucid. This research area would be of great interest in the context of intensional markup. There are many similarities between the variant structure principle in intensional versioning and the inheritance and polymorphism features in OO languages. With the experience and result of the research work in ICC presented in this thesis, inten-

sional OO languages such as intensional C++ or intensional JavaScript or intensional Java inevitably become more promising.

Chapter 6

Conclusions

The main purpose of this thesis is to ease the intensional markup authoring, achieved by designing and implementing tools such as IMP and ICC. IMP follows a sound design pattern and incorporates a JavaScript engine, which greatly enhances the functionality of IML. The intensional spreadsheet approach described in this thesis could present a workable solution for many intensional markup applications such as the French Sentences Maker. This area deserves more attention. ICC provides an alternative to ISE for intensional markup authoring. ICC, it should be noted, is also the first attempt to implement intensional versioning at a compiler level. At present, much experience has been gained which is valuable for future work.

Bibliography

[1] W. W. Wadge, "Intensional markup language," in *Distributed Communities on the Web, Third International Workshop, DCW 2000, Quebec City, Canada, June 19-21, 2000, Proceedings*, ser. Lecture Notes in Computer Science, P. G. Kropf, G. Babin, J. Plaice, and H. Unger, Eds., vol. 1830. Springer, 2000.

[2] P. Swoboda, "Practical languages for intensional programming," M. Sc. Thesis, University of Victoria, 1999.

[3] Home of groff (gnu troff). [Online]. Available: http://www.gnu.org/software/groff/

[4] W. W. Wadge and C. Wadge, "Multidimensional french," in *Proceedings of the Eleventh Annual International Symposium on Languages for Intensional Programming, May 1998, Palo Alto, California, USA*, W. W. Wadge, Ed., pp. 109–117.

[5] W. W. Wadge, "Intensional logic in context," in *Intensional Programming II*, M. Gergatsoulis and P. Rondogiannis, Eds. World Scientific, 2000.

[6] P. Rondogiannis and W. Wadge, "Intensional programming languages." [Online]. Available: http://citeseer.ist.psu.edu/rondogiannis98intensional.html

[7] W. W. Wadge and E. A. Ashcroft, *Lucid, the Dataflow Programming Language*. Academic Press Inc. (London) Ltd., 1985.

[8] J. Plaice and J. Paquet, "Introduction to intensional programming," in *Intensional Programming I*, M. A. Orgun and E. A. Ashcroft, Eds. World Scientific, 1995.

[9] R. Jagannathan and C. Dodd, "Glu programmer's guide (version 0.9)," Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, California 94025, U.S.A., Tech. Rep. SRI-CSL-94-06, 1994. [Online]. Available: http://citeseer.ist.psu.edu/122282.html

[10] R. Jagannathan, "Coarse-grain dataflow programming of conventional parallel computers," in *Advanced Topics in Dataflow Computing and Multithreading*, G. G. E. L. Bic, J-L. Gaudiot, Ed. IEEE Computer Society Press, 1995, pp. 113–129. [Online]. Available: http://citeseer.ist.psu.edu/jagannathan95coarsegrain.html

[11] M. A. Orgun and W. W. Wadge, "Chronolog admits a complete proof procedure," in *the Sixth International Symposium on Lucid and Intensional Programming*, 1993, pp. 120–135. [Online]. Available: http://citeseer.ist.psu.edu/orgun93chronolog.html

[12] J. Plaice and P. G. Kropf, "Intensional communities," in *Intensional Programming II*, M. Gergatsoulis and P. Rondogiannis, Eds. World Scientific, 2000.

[13] B. Wadge and A. Yoder, "The possible-world wide web," in *Intensional Programming I*, M. A. Orgun and E. A. Ashcroft, Eds. World Scientific, 1995.

[14] T. Yildirim, "Intensional html," M. Sc. Thesis, University of Victoria, 1997.

[15] G. D. Brown, "Intensional html 2 : A practical approach," M. Sc. Thesis, University of Victoria, 1998.

[16] Y. Stavrakas, M. Gergatsoulis, and P. Rondogiannis, "Multidimensional xml (extended abstract)." [Online]. Available: http://citeseer.ist.psu.edu/527314.html

[17] M. Gergatsoulis, Y. Stavrakas, and D. Karteris, "Incorporating dimensions in XML and DTD," *Lecture Notes in Computer Science*, vol. 2113, 2001. [Online]. Available: http://citeseer.ist.psu.edu/532736.html

[18] M. Gergatsoulis, Y. Stavrakas, D. Karteris, A. Mouzaki, and D. Sterpis, "A Web-based system for handling multidimensional information through MXML," *Lecture Notes in Computer Science*, vol. 2151, pp. 352–??, 2001. [Online]. Available: http://citeseer.ist.psu.edu/gergatsoulis01webbased.html

[19] V. T. Phong, "Intensional xml," M. Sc. Thesis, University of Victoria, 2000.

[20] J. Plaice and W. W. Wadge, "A new approach to version control," *Software Engineering*, vol. 19, no. 3, pp. 268–276, 1993. [Online]. Available: http://citeseer.ist.psu.edu/plaice93new.html

[21] A. J. Cole, *Macro Processors*. Cambridge University Press, 1981.

[22] B. W. Kernighan and D. M. Ritchie, "The m4 macro processor," *sc unix Programmer s Manual*, vol. 2, 1979. [Online]. Available: http://web.mit.edu/afs/athena.mit.edu/astaff/project/docsourc/doc/unix.manual.progsupp1/17.m4/m4.text

[23] Dennis ritchie home page. [Online]. Available: http://www.cs.bell-labs.com/who/dmr/

[24] Trac programming language. [Online]. Available: http://en.wikipedia.org/wiki/TRAC_programming_language

[25] C. Strachey, "A general purpose macrogenerator," *The Computer Journal*, vol. 8, pp. 225–241, 1965.

[26] P. J. Brown, *Macro Processors and Techniques for Portable Software*. Wiley, January 1974. [Online]. Available: http://www.cs.ukc.ac.uk/pubs/1974/407

[27] Home page of gema. [Online]. Available: http://gema.sourceforge.net/

[28] Xsl transformations (xslt). [Online]. Available: http://www.w3.org/TR/xslt

[29] Home page of cpia. [Online]. Available: http://cpia.sourceforge.net/

[30] Home page of webcompile. [Online]. Available: http://larabell.best.vwh.net/webcompile.html

[31] htmlpp - the html preprocessor. [Online]. Available: http://www.imatix.com/html/htmlpp/index.htm

[32] Home page of mp4h. [Online]. Available: http://packages.debian.org/stable/web/mp4h

[33] W3c document object model. [Online]. Available: http://www.w3.org/DOM/

[34] W. Du and W. W. Wadge, "A 3d spreadsheet based on intensional logic," *IEEE Software*, pp. 78–89, May 1990. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/52.55232

[35] C. W. Fraser and D. R. Hanson, *A Retargetable C Compiler: Design and Implementation*. The Benjamin/Cummings Publishing Company, Inc.

[36] B. W. Kernighan and D. M. Ritchie, *The C Programming Language (Second Edition)*. Prentice Hall.

[37] B. Wadge, "Possible woorlds," in *Intensional Programming I*, M. A. Orgun and E. A. Ashcroft, Eds. World Scientific, 1995.

[38] Nedit. [Online]. Available: http://www.nedit.org/

[39] Crimson editor. [Online]. Available: http://www.crimsoneditor.com/

Appendix A

IMP Reference Manual

A.1 IMP XML Interface

The following summarizes the IMP XML interface. For a detailed description, please see Chapter 3.2.5.

```
<!--
*************************************************
    Macro Definitions

    There are four kinds of macro definitions:

    1. block

       Defines a block which contains normal
       text or other blocks.

    2. ilblock

       Defines an inline block which contains
       normal text or other blocks.

    3. script

       Defines a JavaScript block.

    4. ilscript

       Defines an inline JavaScript block.

    id  :   Macro name.

    fn  :   File name.
```

```
    ln  :   Line number.
************************************************
-->

<!-- block macro definition -->
<!ELEMENT block     (#PCDATA | argc | argv | ref | ilscript
                    | script | ilblock | block)*>
<!ATTLIST block id  CDATA #REQUIRED>

<!-- inline block macro definition -->
<!ELEMENT ilblock   (#PCDATA | argc | argv | ref | ilscript
                    | script | ilblock | block)*>
<!ATTLIST ilblock id CDATA #REQUIRED>

<!-- JavaScript macro definition -->
<!ELEMENT script    (#PCDATA)>
<!ATTLIST script id CDATA #REQUIRED>
<!ATTLIST script fn CDATA #REQUIRED>
<!ATTLIST script ln CDATA #REQUIRED>

<!-- inline JavaScript macro definition -->
<!ELEMENT ilscript  (#PCDATA)>
<!ATTLIST ilscript id CDATA #REQUIRED>
<!ATTLIST ilscript fn CDATA #REQUIRED>
<!ATTLIST ilscript ln CDATA #REQUIRED>


<!--
************************************************
    Macro References

    id  :   Macro name.

    epf :   Indicates the parameters passed into
            the macro would be evaluated at the
            time of macro calling or at the time
            when arguments are requested. Valid
            values are "true" or "false".
************************************************
-->

<!-- reference to macros -->
<!ELEMENT ref       (param)*>
<!ATTLIST ref id    CDATA #REQUIRED>
<!ATTLIST ref epf   CDATA #REQUIRED>

<!--
************************************************
    Parameter Definitions
************************************************
```

```
-->

<!-- parameter definition -->
<!ELEMENT param      (#PCDATA | argc | argv | ref | ilscript
                     | script | ilblock | block)*>

<!--
************************************************
    Parameter References

    id  :   Index of the argument.
************************************************
-->
<!-- reference to the number of arguments -->
<!ELEMENT argc       EMPTY>

<!-- reference to arguments -->
<!ELEMENT argv       EMPTY>
<!ATTLIST argv i     CDATA #REQUIRED>
```

A.2 An IMP Processing Sample

This sample illustrates the processing of an IMP source with the internal data structure passed into the back-end in an XML format and the final output given.

The following is the IMP source:

```
${
    $<
    ******************************************************
        $eol()
    ******************************************************
    $>
    $eol = ${
$[
$]
    $}

    $<
    ******************************************************
        $for(init, condition, update, body)
    ******************************************************
    $>
    $for = $(
        var r = "";
```

```
    bargv(0);
    for (; bargv(1)=="true"; bargv(2)) {
        r += bargv(3);
    }
    r;
$)

$<
*********************************************************
    $dashline()
*********************************************************
$>
$dashline = ${
    $for(
        $( var i=0 $),
        $( i < 20 $),
        $( i++ $),
        $[*$]
    )
    $eol()
$}

$<
*********************************************************
    $m1()
*********************************************************
$>
$m1 = ${
    $m2 = ${
        $[  Hello $]
    $}
    $m3 = ${
        $[world!$]
    $}
$}

$<
*********************************************************
    Hello world!
*********************************************************
$>
$dashline()

$root.m1.m2()

$(
    var r="";
    r = doc.i.root.m1.m3.expand();
    r;
$)
```

```
    $eol()

    $dashline()
$}
```

The front-end translates the above IMP source into the following internal data structure:

```
<ilblock id="root">
    <block id="eol">
        <![CDATA[
]]>
    </block>
    <script id="for" fn="sample.imp" ln="17">
        <![CDATA[
            var r = "";
            bargv(0);
            for (; bargv(1)=="true"; bargv(2)) {
                r += bargv(3);
            }
            r;
        ]]>
    </script>
    <block id="dashline">
        <ref rid="for" epf="0" fn="sample.imp" ln="33">
            <param>
                <ilscript id="$block" fn="sample.imp" ln="33">
                    <![CDATA[ var i=0 ]]>
                </ilscript>
            </param>
            <param>
                <ilscript id="$block" fn="sample.imp" ln="34">
                    <![CDATA[ i < 20 ]]>
                </ilscript>
            </param>
            <param>
                <ilscript id="$block" fn="sample.imp" ln="35">
                    <![CDATA[ i++ ]]>
                </ilscript>
            </param>
            <param>
                <![CDATA[*]]>
            </param>
        </ref>
    <ref rid="eol" epf="0" fn="sample.imp" ln="38"/>
    </block>
    <block id="m1">
        <block id="m2">
            <![CDATA[  Hello ]]>
```

```
        </block>
        <block id="m3">
            <![CDATA[world!]]>
        </block>
    </block>
    <ref rid="dashline" epf="0" fn="sample.imp" ln="60"/>
    <ref rid="root.m1.m2" epf="0" fn="sample.imp" ln="62"/>
    <ilscript id="$block" fn="sample.imp" ln="64">
        <![CDATA[
            var r="";
            r = doc.i.root.m1.m3.expand();
            r;
        ]]>
    </ilscript>
    <ref rid="eol" epf="0" fn="sample.imp" ln="70"/>
    <ref rid="dashline" epf="0" fn="sample.imp" ln="72"/>
</ilblock>
```

After macro processing, the back-end generates the following output:

```
********************
  Hello world!
********************
```

A.3 IMP Syntax Aware Editors and Configuration Scripts

There are many proficient general-purpose text editors and source code editors. Only two are presented here, that of Nedit [38] and Crimson Editor [39]. One common feature of these two WYSIWYG editors is the capability to highlight user-defined syntax.

NEdit is a general-purpose text editor for the X Window system with a graphical user interface and is primarily intended to run on Unix/Linux (although Win32 port and MAC OSX port are also available). The following gives a sample configuration file of IMP syntax for NEdit under Linux, while also defining a few stoke macros to automatically type '${', '$}', '$[', '$]' and '$(', '$)' pairs.

```
nedit.highlightPatterns: \
IMP:1:0{\n\
    A_comment:"\\$\\<":"\\$\\>"::Comment::\n\
```

```
    A_script:"\\$\\(":"\\$\\)"::Preprocessor::\n\
    A_text:"\\$\\[":"\\$\\]"::String::\n\
    A_text_content:"&":"&"::Plain:A_text:C\n\
    A_id:"\\$[a-zA-Z_][a-zA-Z_0-9]*":::Identifier::\n\
    $$_in_comment:"\\$(\\$)":::Comment:A_comment:\n\
    $$_in_script:"\\$(\\$)":::Preprocessor:A_script:\n\
    $$_in_text:"\\$(\\$)":::String:A_text:\n\
    $$_in_comment_2nd$:"\\1":"":Warning:$$_in_comment:C\n\
    $$_in_script_2nd$:"\\1":"":Warning:$$_in_script:C\n\
    $$_in_text_2nd$:"\\1":"":Warning:$$_in_text:C\n\
}


nedit.macroCommands: \
${$} block:Alt+B::: {\n\
    insert_string("${$}")\n\
    backward_character()\n\
    backward_character()\n\
    newline_and_indent()\n\
    beginning_of_line()\n\
    backward_character()\n\
    newline_and_indent()\n\
    insert_string("\\t")\n\
}\n\
$[$] text:Alt+T::: {\n\
    insert_string("$[$]")\n\
    backward_character()\n\
    backward_character()\n\
}\n\
$($) script:Alt+S::: {\n\
    insert_string("$($)")\n\
    backward_character()\n\
    backward_character()\n\
    newline_and_indent()\n\
    beginning_of_line()\n\
    backward_character()\n\
    newline_and_indent()\n\
    insert_string("\\t")\n\
}\n\
```

Figure A.1 is a screen snapshot showing the editing of an IMP program under NEdit for Linux.

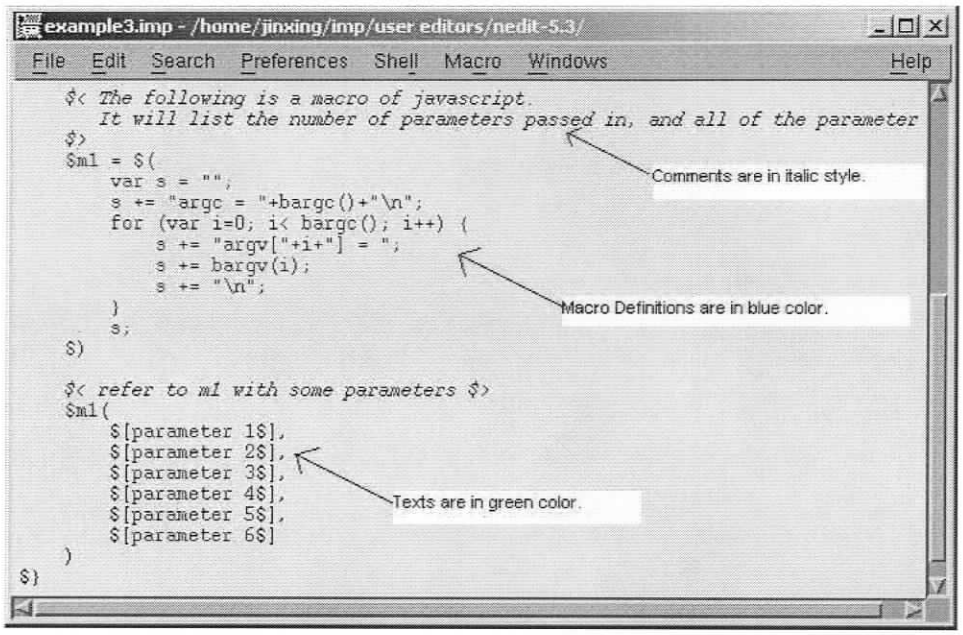


Figure A.1. *IMP Syntax Highlighted In NEdit*

Crimson Editor is a professional source code editor for Windows with a plentiful feature list. The following gives a sample configuration file of IMP syntax for Crimson Editor.

```
# IMP LANGUAGE SPECIFICATION FILE
# FOR CRIMSON EDITOR

$CASESENSITIVE=YES
$VARIABLEPREFIX=$
$SPECIALVARIABLECHARS=$
$BLOCKCOMMENTON=$<
$BLOCKCOMMENTOFF=$>
$RANGE1BEG=$[
$RANGE1END=$]
$RANGE2BEG=$(
$RANGE2END=$)
$INDENTATIONON={
$INDENTATIONOFF=}
$PAIRS1=()
$PAIRS2=[]
$PAIRS3={}
```

Figure A.2 is a screen snapshot showing the editing of an IMP program under Crimson

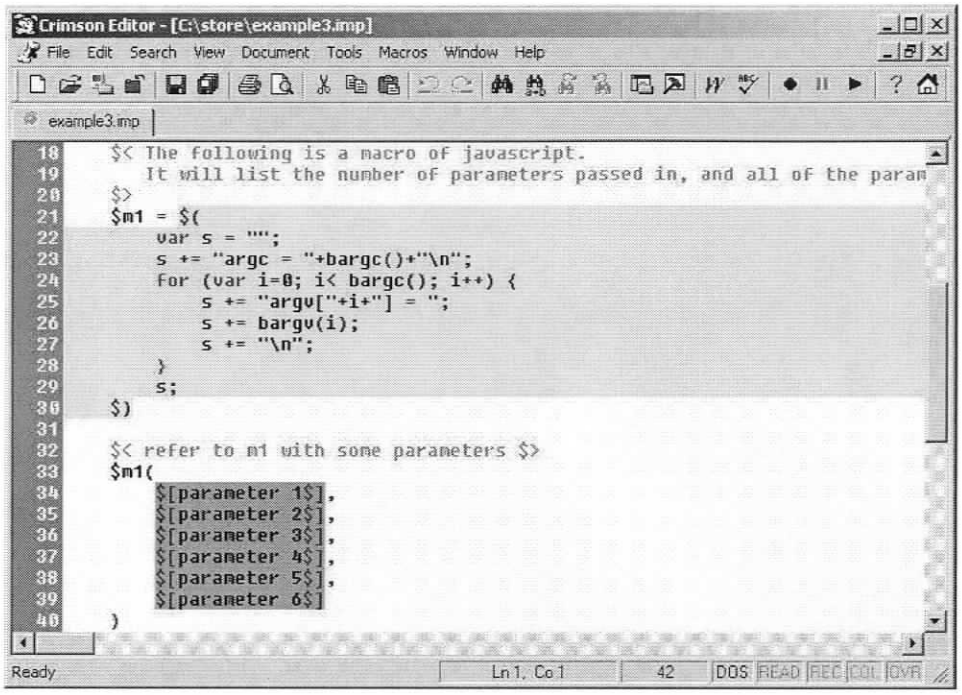


Figure A.2. *IMP Syntax Highlighted In Crimson Editor*

Editor.

A.4 Installation

The installation of IMP is particularly simple. Merely copy the IMP executable file and assign it an executable permission.

To execute IMP, simply issue the 'imp' command with an input file name as the parameter, as shown below:

```
% imp my_imp_script.imp > output.txt
```

Execute IMP without any parameters will result in a simple usage help:

```
% imp
Usage: imp file
```

Appendix B

ICC Reference Manual

The motivations for the creation of ICC and the intensional C language have been described in Chapter 4. This appendix describes the design and implementation of ICC, while offering some pertinent examples and applications of ICC.

B.1 Design

B.1.1 Design Goals

The following are the main design goals of the intensional C language and ICC compiler:

- Compatible - The intensional C language is designed to be a superset of ANSI C. The syntax of intensional C is an extension to ANSI C and maintains the original ANSI C syntax as much as possible. ICC can compile any ANSI C programs in addition to intensional C programs. This backwards compatibility with ANSI C will benefit the C programmers and ease the migration from legacy C codes to intensional C codes.
- Intensional - The intensional C language incorporates the idea of intensional versioning. Intensional identifiers and intensional control constructs are implemented in ICC. These additions would be easily accepted for C programmers.
- Efficient - The compiled executable should run at a fairly acceptable speed. ICC utilizes a selectable intensional identifier scheme. Programmers can decide whether an identifier is intensional to reduce the overload introduced by intensional versioning.

The following were not concerns by this thesis in the initial version of ICC. If one desires to improve ICC one would start with the following areas:

- Intensional types.
- Super speed optimization.
- Super size optimization.
- Cross platform support.

The following subsections describe some of ICC's key design ideas.

B.1.2 Intensional Versioning Scheme

The version language scheme used in ICC is shown in the following context-free grammar and regular expression:

$$
\begin{aligned}
\textit{version\_string} &\longrightarrow \\
&\longrightarrow \textit{dimension\_list} \\
\textit{dimension\_list} &\longrightarrow \textit{dimension\_list\_element dimension\_list\_rest} \\
\textit{dimension\_list\_rest} &\longrightarrow \\
&\longrightarrow \text{`+'}\ \textit{dimension\_list} \\
\textit{dimension\_list\_element} &\longrightarrow \textit{DIMVAL} \\
&\longrightarrow \textit{DIMNAME version} \\
\textit{version} &\longrightarrow \\
&\longrightarrow \textit{DIMVAL} \\
&\longrightarrow \textit{DIMNAME version} \\
&\longrightarrow \text{`('}\ \textit{dimension\_list}\ \text{`)'}
\end{aligned}
$$

$$
\begin{aligned}
\textit{DIMVAL} &:= \textit{[0-9a-zA-Z\_]+} \\
\textit{DIMNAME} &:= \textit{([0-9a-zA-Z\_]+:)+}
\end{aligned}
$$

This version scheme differs from the scheme used in ISE in that a version string can be empty to denote a ε version; values such as '$\sim$', '$\wedge$', and '?', which denote minimum version, maximum version and requested version respectively as in ISE, are not allowed in ICC. Being a simplified version of the scheme described in [15] and [2], this version scheme is practical for the initial version of ICC. Future versions of ICC might consider implementations of a full version scheme.

B.1.3 Intensional Version Manipulation

To manipulate the intensional versioning scheme, a set of intensional version operators must be defined. Both IHTML and ISE did much work in this area. To further this idea, a standardized operator library should be taken into consideration. A standard operator library will bring a great benefit to the future research in intensional programming, considering that more tools for intensional versioning will invariably be invented in the near future. The followings are some operators that could be considered:

- Given a version, an intensional version parser could tell if it is a valid version form.
- Given a version, a canonical form representation of the version can be returned.
- Given a version, a factored canonical form representation of the version can be returned.
- Given a version, a string representation can be returned.
- Given a version, the number of items can be retrieved.
- Given a version and a DIMNAME, it would be possible to deduce whether the DIMNAME is in the version.
- Given a version and a DIMNAME, the DIMVAL of the DIMNAME in the version can be returned or set.
- Given two versions, a refinement test can deduce if the first version refines to the second and if the second refines to the first.

- Given two versions, a new version can be created by concatenating the two versions.
- Given two versions, a new version can be created by removing each DIMNAME item appearing in the second version from the first version.
- Given two versions, one version can be modified by the other version. This is referred as 'VMOD' in [15].
- Given a set of versions and a requested version, a best-fit test can tell if there is a best-fit version, and the best-fit version can be retrieved if it exists.

In addition to these operators for intensional versioning, the following operators at an application level could also be considered:

- Providing facilities to define, request, retrieve and switch contexts. The applications may maintain one or more global contexts.
- Allowing evaluating context at both runtime and compilation time.
- Supporting aggregation in which all versions refined to the current version can be grouped together. This notion of aggregation first appears in [14].
- Maintaining a hit context history. A hit context history records the context switch history and will help to analyze the context walking paths.
- Easing the binding operation of the versions and the application objects.

Ideally, the operators can be encapsulated in C++ classes akin to what ISE did in some of the cases of the operators previously mentioned. In the initial version of ICC many of the aforementioned operators have been implemented in C.

B.1.4 Selectable Intensional Identifier Scheme

Paul Swoboda in [2] observes:

Because of the computational overhead associated with intensional best-fits

on every variable-access and function-call, it would make little sense to implement such a tongue in the form of a compiler...

The above statement is true is mainly because of the following two reasons: Firstly, intensional languages are traditionally designed to treat every identifier as an intensional entity. In Lucid, all variables denote streams or sequences of data items [7]. In ISE, versioning are applied to all of the identifiers, including variables, files, and functions [2]. Secondly, performance is always a critical concern in intensional programming. To perform best-fit on every variable access and function-call would result a great slow speed at runtime.

To avoid performance penalty, declarative intensional languages such as Lucid take an Education approach as the basic computation model. Education is a tagged demand-driven computation model in which operations are driven by demands for the results of these operations and their contexts. Imperative interpreted languages such as ISE take the same approach. Unfortunately, this approach is not ideal for designing intensional C at a compiler level for the reasons described.

Instead of Education, ICC takes a selectable intensional identifier scheme. In reality, this becomes a trade-off between the intensional scope and the performance, allowing programmers to make decisions for identifiers whether they are intensional. By doing this, the ICC compiler can produce normal codes for non-intensional identifier accesses and produce additional codes for intensional identifier accesses.

B.1.5 Superset of ANSI C

ICC is designed to be a superset of ANSI C. Most of the syntax and semantic meanings of ANSI C are preserved while some are changed:

- Variables and function names can be selected to be intensional. This is a key feature that ICC extends from ANSI C.
- Constants are used in the same manner as in ANSI C. Example of constants are

123, 'x', '123.4' and "a string". It is meaningless for constants to have intensional meanings. Constants should not be confused with a const qualifier.

- Labels can be selected to be intensional. Although labels are not recommended to appear in well-styled program structures, they sometimes can be useful.
- Types are used in the same manner as in ANSI C. They are not intensionally versioned. Intensional versioned types could be an interesting subject for intensional programming language researchers because it is still unclear what semantic meanings intensional types should have.
- Statements now include a new member, 'vswitch'. Other statements are similar to ANSI C.
- Scope now possesses a few more rules. An intensional variable name and a non-intensional identifier name cannot share the same name in the same scope. A intensional function name cannot share the same name as a non-intensional function in the same module. A intensional label name cannot share the same name as a non-intensional label in the same function.
- Structs and unions are used in the same manner as in ANSI C. But variables of structs and unions can be used in an intensional way.
- Enumerations are used in the same manner as in ANSI C.
- Storage class and linkage types are the same as in ANSI C.

B.2 Implementation

This section covers some important issues on ICC implementation. For more details of the implementation, please refer to the ICC source code package.

B.2.1 Approaches

One approach to implement ICC is to build a completely new compiler from zero. The benefit is that the lexical analyzer, the syntax parser, and the semantic analyzer parts are relatively simple to write. The deficiency is the code optimization with the code generation parts performing the bulk of the work. Another approach is to make ICC from an existing ANSI C compiler. A beneficial aspect is the code optimization and the code generation parts are already performed, while the negative aspect is that the need to extend the lexical analyzer, the syntax parser, and the semantic analyzer is even more difficult than to write them from the ground up. This is even worse if one considers the source code size for such a software such as a compiler and the complexity of the source code that is usually optimized for better performance for practical reasons.

The initial version of ICC extends an existing compiler, mainly because the author of this thesis was reluctant to devote energy into the tedious testing for the code optimization and code generation parts.

Two existing compilers were taken into consideration, GNU's GCC and the LCC made at Princeton University. Both of the compilers follow the modern compiler design that explicitly separates the front-end phase from the back-end phase. The author of this thesis chose LCC because of its better documentations. The initial version of ICC is based on the LCC version 4.2.

B.2.2 'Refine To' Algorithm

The following describes the 'refines to' algorithm implemented in ICC which one should note is not precisely the same as in ISE.

```
refines_to (V1, V2)
{
    if (V1 == V2) {
        return TRUE;
```

```
    } else if (V1 == ε) {
        return TRUE;
    } else if (V2 == ε) {
        return FALSE;
    }
    foreach di ∈ D(V1) {
        if ( γ(di, V2) == ε) {
            return FALSE;
        }
        if ( (γ(di, V1), γ(di, V2) ∈ Z) && (γ(di, V1) > γ(di, V2))) {
            return FALSE;
        }
        if ( γ(di, V1) != γ(di, V2)) {
            return FALSE;
        }
    }
    return TRUE;
}
```

Where V represents a version, ε represents *vanilla* version, d represents a dimension name, $D(V)$ represents the set of all dimension names in version V and $\gamma(d, V)$ represents the value of dimension d in version V.

B.2.3 'Best Fit' Algorithm

The following describes the 'best fit' algorithm implemented in ICC, which is the same as that found in ISE.

```
best_fit (Vreq, S)
{
```

```
if (V_req ∈ S) {
    return V_req;
}
V_best = NULL;
S_2 = φ;
foreach V_i ∈ S {
    if ( V_i ⊆ V_req) {
        if ( V_best ⊆ V_i) {
            V_btest = V_i;
        } else if ( V_i ⊈ V_best) {
            S_2 = S_2 ∪ V_i;
        }
    }
}
foreach V_i ∈ S_2 {
    if ( V_i ⊈ V_best) {
        return NULL;
    }
}
return V_best;
}
```

Where V represents a version, S represents a set of versions, and $V_1 \subseteq V_2$ represents V_1 refines to V_2.

B.2.4 Static Version and Dynamic Version

A version string can be a constant string such as @lg:en@. The value of the string can be determined at compile time, which is referred to as static version.

A version string can also be a string expression which contains other variables such as @'str1'+'str2'@ where 'str1' and 'str2' are variables. The value of the string cannot be determined at compile time but at runtime, which is referred to as dynamic version.

The initial version of ICC only implements the static version. Future versions of ICC may implement the dynamic version.

As a comparison with ISE, ISE supports both static version and dynamic version for variables. Only the static version for function names rather than dynamic version is supported in ISE.

B.2.5 Built-in or Standard Library

ICC maintains only one global context for one runtime. 'char * _icc_rt_version' is used to refer to this global context. In addition to the changes in grammar, ICC also contains many of the context manipulating functions described in Chapter B.1.3.

One problem is to determine where to place all of these extra global variables and functions. ICC has placed some frequently used global variables and functions into built-in tables that are automatically inserted to the head of each C file compiled. The ICC programmers can use these names directly without declaring or including anything. ICC also has positioned some declarations of global variables and functions into an .H file and implemented them in a .C file that is automatically linked as standard library at linking time.

B.2.6 Built-in Variables and Functions

The following variable is used quite frequently making it is better for programmers to use it without declaration:

```
extern char * _icc_rt_version;
```

'_icc_rt_version' refers to the single global context maintained by ICC runtime. Programmers can directly use this varaible as shown in the examples of Chapter B.3.

B.2.7 Variables and Functions as Standard Library

The following functions are related to the intensional version operations described in Chapter B.1.3. Programmers need to include 'ivl.h' in order to use them.

```
extern int     _icc_vparse(char *vbuf);
```

The above function parses a string to see whether it conforms to the version grammar described in Chapter B.1.2.

```
extern char * _icc_vnormalize(char *v);
```

The above function standardizes a version string to its canonical form.

```
extern int     _icc_vrefinesto(char *v1, char *v2);
```

The above function tests if version v1 refines to version v2.

```
extern _icc_VsP _icc_vbestfit(char *req, _icc_VsP head);
```

The above function resolves the best fit version of a requested version in a version space.

```
extern char * _icc_vgetdimval(char * v, char * dimname);
```

The above function returns a dimension value by a version and a dimension name.

```
extern char * _icc_vmod(char * v, char * mod);
```

The above function applies a VMOD operation to a version.

B.2.8 Lexical Analysis

A lexical analyzer reads source text and produces tokens that are the basic lexical units of the language.

One problem is choosing a delimiter for version strings. To avoid conflict with ANSI C operators and keywords, only '@', '$' and '/' are available. '$' and '/' would generate problems in the generated assembler code, making '@' is the only acceptable choice.

Two tokens are added into the token table shown as below:

```
xx(symbol,  value, prec, op, optree, kind, string)

xx(VSWITCH, 128,   0,    0,  0,      IF,   "vswitch")
xx(VID,     129,   0,    0,  0,      0,    "version identifier")
```

'VSWITCH' identifies 'vswitch' keyword, while 'VID' represents version string.

B.2.9 Syntactical and Semantic Analysis

A syntactical parser confirms that the input tokens conforms to the syntax of the language and builds an internal representation of the input source program.

Symbol tables are also built at this time. Two fields, 'ivl_mv' and 'ivl_vs' are added in the symbol structure shown below:

```
struct symbol {
    char *name;
    int scope;
    Coordinate src;
    Symbol up;
    List uses;
    int sclass;
    unsigned structarg:1;

    unsigned addressed:1;
    unsigned computed:1;
    unsigned temporary:1;
    unsigned generated:1;
    unsigned defined:1;
    Type type;
    float ref;
```

```
    union {
        struct {
            int label;
            Symbol equatedto;
        } l;
        struct {
            unsigned cfields:1;
            unsigned vfields:1;
            Table ftab;     /* omit */
            Field flist;
        } s;
        int value;
        Symbol *idlist;
        struct {
            Value min, max;
        } limits;
        struct {
            Value v;
            Symbol loc;
        } c;
        struct {
            Coordinate pt;
            int label;
            int ncalls;
            Symbol *callee;
        } f;
        int seg;
        Symbol alias;
        struct {
            Node cse;
            int replace;
            Symbol next;
        } t;
    } u;
    Xsymbol x;
    int   ivl_mv;  /* true if this symbol is intensional */
    void *ivl_vs;  /* pointer to version space if ivl_mv is true */
};
```

The version space is a data structure that keeps record of all the possible versions of the symbol. In ICC's initial implementation, this data structure is designed to be a linked list. Future versions of ICC could be designed in data structures for better space or better speed. ISE [2] uses a threaded AVL tree data structure to achieve better space. The following shows the version space of ICC in a linked list form:

```
struct _icc_vs {
    char * version; /* point to version name string */
    _icc_VsP next;  /* next version in the version space */
    void * pdata;   /* bound object */
};
```

The semantic analysis is the most difficult part to implement, and is scattered among many locations inside several source files. The function 'static Tree primary(void)' in file expr.c is where all identifiers can be intensional versioned. The function 'void statement(int loop, Swtch swp, int lev)' in file stmt.c is where 'vswitch' statement is implemented. The function 'static Symbol dclglobal(int sclass, char *id, Type ty, Coordinate *pos)' in file decl.c and several other functions in the same file have various semantic checkings added. For details of the implementation, please refer to the ICC source code package.

B.2.10 Other Phases

To use the ICC compiler, any C preprocessor can be used directly. The C preprocessors from GCC and LCC provide good examples to be used in conjunction with ICC. The lexical and the syntactical additions of intensional C language do not have any problem with the C preprocessors.

When generating a pure ANSI C source code, the generated pseudo-code by ICC may be somewhat different from that of the LCC at binary level, although the semantic meanings are identical, as the IR tree generated by ICC is slightly different than that of LCC.

B.2.11 Debugging and Profiling

One of the advantages of ICC over ISE is that ICC is compatible with common debugging tools such as GNU's GDB. Although GDB is not an ideal debugging tool for intensional programmers, as it cannot provide context sensitive debugging facilities such as setting intensional breakpoints, debugging in ICC is better than that in ISE considering the poor debugging facilities coming along with ISE.

To include debugging information, use the '-g' switch shown below:

```
%icc -g -o fd.o fd.c
```

GDB, in addition to other current available debuggers, do not understand the intensional meaning of the programs debugged. Breakpoints and watches cannot be set to intensional functions nor can variables be set, but one may set to individual versions of functions or variables, as shown in the following example:

```
%gdb fd.o
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are welcome to change it and/or distribute copies of it under certain
conditions. Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...

(gdb) l 1, 30
1       #include <stdio.h>
2
3       int sum@d:1+g:1@;
4       int sum@d:2@;
5
6       void f1@d:1+g:1@() {
7           printf("f1@d:1+g:1@ executed.\n");
8       }
9
10      void f1@d:2@() {
11          printf("f1@d:2@ executed.\n");
12      }
13
14      void main(int argc, char * argv[]) {
15          if (argc > 1) {
16              int i;
17              for (i=1; i<argc; i++)
18                  if (argv[i][0] == '-' && argv[i][1] == 'v')
19                      if (_icc_vparse(&argv[i][2])==0)
20                          _icc_rt_version = _icc_vnormalize(&argv[i][2]);
21          }
22
23          sum = 1;
24          f1();
25      }
```

```
(gdb) break 'f1.d$1.g$1.'
Breakpoint 1 at 0x8048640: file fd.c, line 6.

(gdb) run -vd:1+g:1
Starting program: /home/jinxing/icc/fd.o -vd:1+g:1

Breakpoint 1, f1.d$1.g$1. () at fd.c:6
6       void f1@d:1+g:1@() {

(gdb) print 'sum.d$1.g$1.'
$1 = 1

(gdb)
```

In the above example, symbol 'f1@d:1+g:1@' is changed into 'f1.d$1.g$1.' in GDB and must be quoted by '/' as the delimiters of the version string have been changed by the ICC compiler to avoid problems in the generated assembler code. The rules have '@' and '+' changed into '.' and ':' changed into '$'.

Profiling is another advantage of ICC over ISE. The profiling facilities of ICC are inherited from LCC. There are two methods to profile an ICC program, one is based on the bprint utility coming with LCC and the other is based on GNU's Gprof.

The following example shows the use of bprint utility to profile an ICC program. It is a frequency-based profiling that records the number of times each expression executed. Execution counts are enclosed in angle brackets.

```
% cat sum.c
#include <stdio.h>

int i, sum=0;
int start@@=0;
int start@d:100@ = 1;
int end@@ = 2003;
int end@d:100@ = 100;

void fsum@@()
{
    for (i=start;i<=end;i++) {
        sum += i;
    }
}
```

```
void fsum@m:1@()
{
    sum = (start+end) * (end-start+1) / 2;
}

void main(int argc, char * argv[])
{

    if (argc > 1) {
        int i;
        for (i=1; i<argc; i++)
            if (argv[i][0] == '-' && argv[i][1] == 'v')
                if (_icc_vparse(&argv[i][2])==0)
                    _icc_rt_version = _icc_vnormalize(&argv[i][2]);
    }

    fsum();

    printf("sum(%i..%i)=%i\n", start, end, sum);
}
```

```
% icc -b -o sum.o sum.c
% sum.o -vd:100+m:1
sum(1..100)=5050
```

```
% bprint
#include <stdio.h>

int i, sum=0;
int start@@=0;
int start@d:100@ = 1;
int end@@ = 2003;
int end@d:100@ = 100;

void fsum@@()
<0>{
    for (<0>i=start;<0>i<=end;<0>i++) <0>{
        <0>sum += i;
    <0>}
<0>}

void fsum@m:1@()
<1>{
    <1>sum = (start+end) * (end-start+1) / 2;
<1>}

void main(int argc, char * argv[])
<1>{

    if (<1>argc > 1) <1>{
```

```
        int i;
        for (<1>i=1; <2>i<argc; <1>i++)
            if (<1>argv[i][0] == '-' && <1>argv[i][1] == 'v')
                if (<1>_icc_vparse(&argv[i][2])==0)
                    <1>_icc_rt_version = _icc_vnormalize(&argv[i][2]);
    <1>}

    <1>fsum();

    <1>printf("sum(%i..%i)=%i\n", start, end, sum);
<1>}
```

The following example shows the use of Gprof utility to profile an ICC program, which samples the location counter periodically to obtain an estimate of the percentage of total execution time spent on each function while also reporting the call graph of the functions.

```
% icc -pg -o sum.o sum.c
% sum.o -vd:100+m:1
sum(1..100)=5050

% gprof sum.o
Flat profile:

Each sample counts as 0.00195312 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00        1     0.00     0.00  fsum.m$1.

...


             Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) no time propagated

index % time    self  children    called     name
                0.00    0.00       1/1          main [23]
[1]      0.0    0.00    0.00       1          fsum.m$1. [1]
-----------------------------------------------

...
```

B.3 Examples

B.3.1 Example - HELLO.C

The first ICC example is the famous 'Hello World!' which shows the intensional variables in ICC programming.

```
/*
    hello.c

    Command Line Options:

    -vlg:xx where xx is the short for the following languages:
                    en : English
                    fr : French
                    la : Latin
                    be : Bengali
                    ge : German
                    hi : Hindi
                    it : Italian
                    no : Norwegian
                    po : Portugese

            If there is no -vlg:xx option specified, English
            is the default value.
*/

#include <stdio.h>

int main(int argc, char * argv[])
{
    char * msg@@;
    char * msg@lg:en@ = "Hello world!       \n"; /* English   */
    char * msg@lg:fr@ = "Bonjour Monde!     \n"; /* French    */
    char * msg@lg:la@ = "Vale Mundum!       \n"; /* Latin     */
    char * msg@lg:be@ = "Shagatam Prithivi! \n"; /* Bengali   */
    char * msg@lg:ge@ = "Hallo Welt!        \n"; /* German    */
    char * msg@lg:hi@ = "Shwagata Prithvi!  \n"; /* Hindi     */
    char * msg@lg:it@ = "Ciao Mondo!        \n"; /* Italian   */
    char * msg@lg:no@ = "Hallo verden!      \n"; /* Norwegian */
    char * msg@lg:po@ = "Ola mundo!         \n"; /* Portugese */

    msg@@ = msg@lg:en@;                         /* default   */

    /* get context request from environment */
    if (argc > 1) {
        int i;
```

```
        for (i=1; i<argc; i++)
            if (argv[i][0] == '-' && argv[i][1] == 'v')
                if (_icc_vparse(&argv[i][2])==0)
                    _icc_rt_version = _icc_vnormalize(&argv[i][2]);
    }

    printf("%s", msg);

    return 0;
} /* end of main */
```

Running the above program with a French version request results in the following:

```
% hello -vlg:fr
Bonjour Monde!
```

Running the above program with an unexpected version request results in the following:

```
% hello -vlg:zz
Hello world!
```

Running the above program without any version request results in the following:

```
% hello
Hello world!
```

B.3.2 Example - ESF.C

This example simply prints out words in a sentence. For example, 'the quick brown fox jumps over the lazy dog' appears in a sorted order or in a reversed sorted order, depending on the context request. Thus, this example shows how functions can be written in an intensional manner.

```
/*
    esf.c

    Command Line Options:
```

```
    -vVERSION   where VERSION is any combination of the
        following dimensions and values:

        dimension   value
        ------------------------------------------------
        sort        no      - with no sort(default)
                    bubble  - with bubble sort

        order       normal  - with normal order(default)
                    reverse - with reverse order

*/

#include <stdio.h>
#include <ctype.h>

/* calculate the number of words in the array */
int arrlen(char *a[])
{
    int i, c=0;

    for(i=0; a[i][0]!=0; i++)
        c++;
    return c;
}

/* print the array */
void print(char *a[])
{
    int i;

    for(i=0; a[i][0]!=0; i++)
        printf("%s ", a[i]);
    printf("\n");
}

/* normal sort */
void sort@sort:no@(char *a[])
{
}

/* default sort */
void sort@@(char *a[])
{
    sort@sort:no@(a);
}

/* bubble sort */
void sort@sort:bubble@(char *a[])
{
```

```
    int i, j;
    int size = arrlen(a);

    for (i=0;i<size-1;i++)
        for (j=i+1;j<size;j++)
            if (strcmp(a[i], a[j])>0) {
                char * t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
}

/* normal order */
void order@order:normal@(char *a[])
{
}

/* default order */
void order@@(char *a[])
{
    order@order:normal@(a);
}

/* reverse order */
void order@order:reverse@(char *a[])
{
    int i, j;
    int size = arrlen(a);

    for(i=0; i<size/2; i++) {
        char * t = a[i];
        a[i] = a[size-i-1];
        a[size-i-1] = t;
    }
}

/* main */
void main(int argc, char * argv[])
{
    /* the array */
    char *a[] = { "the", "quick", "brown", "fox",
        "jumps", "over", "the", "lazy", "dog", "" };

    /* get version request from environment */
    if (argc > 1) {
        int i;
        for (i=1; i<argc; i++)
            if (argv[i][0] == '-' && argv[i][1] == 'v')
                if (_icc_vparse(&argv[i][2])==0)
                    _icc_rt_version = _icc_vnormalize(&argv[i][2]);
```

```
    }

    /* sort the array */
    sort(a);

    /* order the array */
    order(a);

    /* print the array */
    print(a);
}
```

The following shows the output of the above program under several different context requests:

```
% esf
the quick brown fox jumps over the lazy dog

% esf -vsort:bubble
brown dog fox jumps lazy over quick the the

% esf -vsort:bubble+order:reverse
the the quick over lazy jumps fox dog brown
```

B.3.3 Example - ESV.C

This example behaves the same as example esf.c, but uses 'vswitch' instead of intensional functions.

```
/*
    esv.c

    Command Line Options:

    -vVERSION   where VERSION is any combination of the
        following dimensions and values:

        dimension   value
        ------------------------------------------------
        sort        no      - with no sort(default)
                    bubble  - with bubble sort

        order       normal  - with normal order(default)
```

```
                    reverse - with reverse order

*/

#include <stdio.h>
#include <ctype.h>

/* calculate the number of words in the array */
int arrlen(char *a[])
{
    int i, c=0;

    for(i=0; a[i][0]!=0; i++)
        c++;
    return c;
}

/* print the array */
void print(char *a[])
{
    int i;

    for(i=0; a[i][0]!=0; i++)
        printf("%s ", a[i]);
    printf("\n");
}

/* sort */
void sort(char *a[])
{
    vswitch {
        case @@ :
        case @sort:no@ :{
                break;
        }
        case @sort:bubble@ : {
                /* bubble sort */
                int i, j;

                for (i=0;i<arrlen(a)-1;i++)
                    for (j=i+1;j<arrlen(a);j++)
                        if (strcmp(a[i], a[j])>0) {
                            char * t = a[i];
                            a[i] = a[j];
                            a[j] = t;
                        }
                break;
        }
    }
}
```

```
/* order */
void order(char *a[])
{
    vswitch {
        case @@ :
        case @order:normal@ : {
                break;
            }
        case @order:reverse@ : {
                int i, j;

                for(i=0; i<arrlen(a)/2; i++) {
                    char * t = a[i];
                    a[i] = a[arrlen(a)-i-1];
                    a[arrlen(a)-i-1] = t;
                }
                break;
            }
    }
}

/* main */
void main(int argc, char * argv[])
{
    /* the array */
    char *a[] = { "the", "quick", "brown", "fox",
        "jumps", "over", "the", "lazy", "dog", "" };

    /* get version request from environment */
    if (argc > 1) {
        int i;
        for (i=1; i<argc; i++)
            if (argv[i][0] == '-' && argv[i][1] == 'v')
                if (_icc_vparse(&argv[i][2])==0)
                    _icc_rt_version = _icc_vnormalize(&argv[i][2]);
    }

    /* sort the array */
    sort(a);

    /* order the array */
    order(a);

    /* print the array */
    print(a);
}
```

The following shows the output of the above program under several different context

requests:

```
% esv
the quick brown fox jumps over the lazy dog

% esv -vsort:bubble
brown dog fox jumps lazy over quick the the

% esv -vsort:bubble+order:reverse
the the quick over lazy jumps fox dog brown
```

B.3.4 Example - ESL.C

This example uses intensional labels to achieve the same result.

```
/*
    esl.c

    Command Line Options:

    -vVERSION   where VERSION is any combination of the
        following dimensions and values:

        dimension   value
        ------------------------------------------------
        sort        no      - with no sort(default)
                    bubble  - with bubble sort

        order       normal  - with normal order(default)
                    reverse - with reverse order

*/

#include <stdio.h>
#include <ctype.h>

/* calculate the number of words in the array */
int arrlen(char *a[])
{
    int i, c=0;

    for(i=0; a[i][0]!=0; i++)
        c++;
    return c;
}
```

```
/* print the array */
void print(char *a[])
{
    int i;

    for(i=0; a[i][0]!=0; i++)
        printf("%s ", a[i]);
    printf("\n");
}

/* sort */
void sort(char *a[])
{
    int i, j;

    goto label;

label@sort:bubble@:
    /* bubble sort */

    for (i=0;i<arrlen(a)-1;i++)
        for (j=i+1;j<arrlen(a);j++)
            if (strcmp(a[i], a[j])>0) {
                char * t = a[i];
                a[i] = a[j];
                a[j] = t;
            }

label@@:
label@sort:no@:
    return;
}

/* order */
void order(char *a[])
{
    int i, j;

    goto label;

label@order:reverse@ :
    for(i=0; i<arrlen(a)/2; i++) {
        char * t = a[i];
        a[i] = a[arrlen(a)-i-1];
        a[arrlen(a)-i-1] = t;
    }

label@@:
label@order:normal@ :
    return;
```

```
}

/* main */
void main(int argc, char * argv[])
{
    /* the array */
    char *a[] = { "the", "quick", "brown", "fox",
        "jumps", "over", "the", "lazy", "dog", "" };

    /* get version request from environment */
    if (argc > 1) {
        int i;
        for (i=1; i<argc; i++)
            if (argv[i][0] == '-' && argv[i][1] == 'v')
                if (_icc_vparse(&argv[i][2])==0)
                    _icc_rt_version = _icc_vnormalize(&argv[i][2]);
    }

    /* sort the array */
    sort(a);

    /* order the array */
    order(a);

    /* print the array */
    print(a);
}
```

The following shows the output of the above program under several different context requests:

```
% esl
the quick brown fox jumps over the lazy dog

% esl -vsort:bubble
brown dog fox jumps lazy over quick the the

% esl -vsort:bubble+order:reverse
the the quick over lazy jumps fox dog brown
```

B.4 Applications

This section illustrates two applications of ICC.

B.4.1 CGI Programming for IHTML

One application area of ICC is IHTML programming in an intensional CGI manner. The concepts of IHTML are described in [14], [15] and [2]. Similar to ISE, ICC provides another useful tool to the family of IHTML programming. Compared with ISE, ICC provides the following features in IHTML programming:

- ICC programming is C based. C is more attractive than its sibling ISE for users who know C/C++ and Java but are reluctant to accept the arcane style of Perl-like ISE grammar.
- The CGI programs written in ICC are compiled into executable CGI programs. They run faster than the equivalent interpreted ISE programs.
- Since the CGI programs written in ICC are real CGI executables, no server side tools are needed. To run a CGI program written in ISE, an ISE interpreter must be installed in the server side and properly configured in the HTTP server.

The following shows a simple example for CGI programming in ICC. This particular example is almost identical to the hello.c in Chapter B.3.1 except that the input and output methods used are different.

```
/*

    hellocgi.c

*/

#include <stdio.h>
#include <stdlib.h>

/* HTML header */
void beginHtml(char * title)
{
    printf("Content-type: text/html\n\n");
    printf("<html>\n<head>\n");
    printf("<title>%s</title>\n", title);
    printf("</head>\n");
}
```

```
/* HTML footer */
void endHtml()
{
    printf("</html>\n");
}

/* context switches */
void print_links()
{
    char * lg = _icc_vgetdimval(_icc_rt_version, "lg");

    printf("Select a version:<br>");
    printf("<a href=\"hello.cgi<lg:en>\">English   </a><br>");
    printf("<a href=\"hello.cgi<lg:fr>\">French    </a><br>");
    printf("<a href=\"hello.cgi<lg:la>\">Latin     </a><br>");
    printf("<a href=\"hello.cgi<lg:be>\">Bengali   </a><br>");
    printf("<a href=\"hello.cgi<lg:ge>\">German    </a><br>");
    printf("<a href=\"hello.cgi<lg:hi>\">Hindi     </a><br>");
    printf("<a href=\"hello.cgi<lg:it>\">Italian   </a><br>");
    printf("<a href=\"hello.cgi<lg:no>\">Norwegian </a><br>");
    printf("<a href=\"hello.cgi<lg:po>\">Portugese </a><br>");
    printf("<a href=\"hello.cgi<lg:ch>\">Chinese   </a><br>");
    printf("<a href=\"hello.cgi<lg:sw>\">Swedish   </a><br>");
    printf("<a href=\"hello.cgi<lg:tu>\">Turkish   </a><br>");
    printf("<a href=\"hello.cgi<lg:sp>\">Spanish   </a><br>");
    printf("<a href=\"hello.cgi<lg:cz>\">Czech     </a><br>");
}

/* say hello world */
void print_hello()
{
    char * msg@@;
    char * msg@lg:en@ = "Hello world!       \n";    /* English   */
    char * msg@lg:fr@ = "Bonjour Monde!     \n";    /* French    */
    char * msg@lg:la@ = "Vale Mundum!       \n";    /* Latin     */
    char * msg@lg:be@ = "Shagatam Prithivi! \n";    /* Bengali   */
    char * msg@lg:ge@ = "Hallo Welt!        \n";    /* German    */
    char * msg@lg:hi@ = "Shwagata Prithvi!  \n";    /* Hindi     */
    char * msg@lg:it@ = "Ciao Mondo!        \n";    /* Italian   */
    char * msg@lg:no@ = "Hallo verden!      \n";    /* Norwegian */
    char * msg@lg:po@ = "Ola mundo!         \n";    /* Portugese */
    char * msg@lg:ch@ = "\xC4\xE3\xBA\xC3   \n";    /* Chinese   */
    char * msg@lg:sw@ = "Hejsan v\x84rld!   \n";    /* Swedish   */
    char * msg@lg:tu@ = "Merhaba D\x81nya!  \n";    /* Turkish   */
    char * msg@lg:sp@ = "\xADHola Mundo!    \n";    /* Spanish   */
    char * msg@lg:cz@ = "Ahoj, sv\x88t!     \n";    /* Czech     */

    msg@@ = msg@lg:en@;                             /* default   */

    printf("<hr>");
```

```
    printf("%s\n", msg);
}

/* HTML body  */
void print_body()
{
    printf("<body>");
    print_links();
    print_hello();
    printf("</body>");
}

/* the main function */
void main()
{
    /* get context request from environment */
    char * reqv = getenv("REQUESTED_VERSION");
    if (reqv && _icc_vparse(reqv)==0)
        _icc_rt_version = _icc_vnormalize(reqv);

    /* the HMTL content */
    beginHtml("Hello World in ICC");
    print_body();
    endHtml();
}
```

'hellocgi.c' obtains its context request from the environment variable 'REQUESTED_VERSION' which is parsed by the HTTP server from the parameters along with the URL requested, whereas 'hello.c' obtains its context request from command line option buffers. 'hello.c' simply prints out messages to the standard output, whereas 'hellocgi.c' has to wrap the messages with HTML syntax and an HTTP header. Figure B.1 shows the screen snapshot of this CGI example.

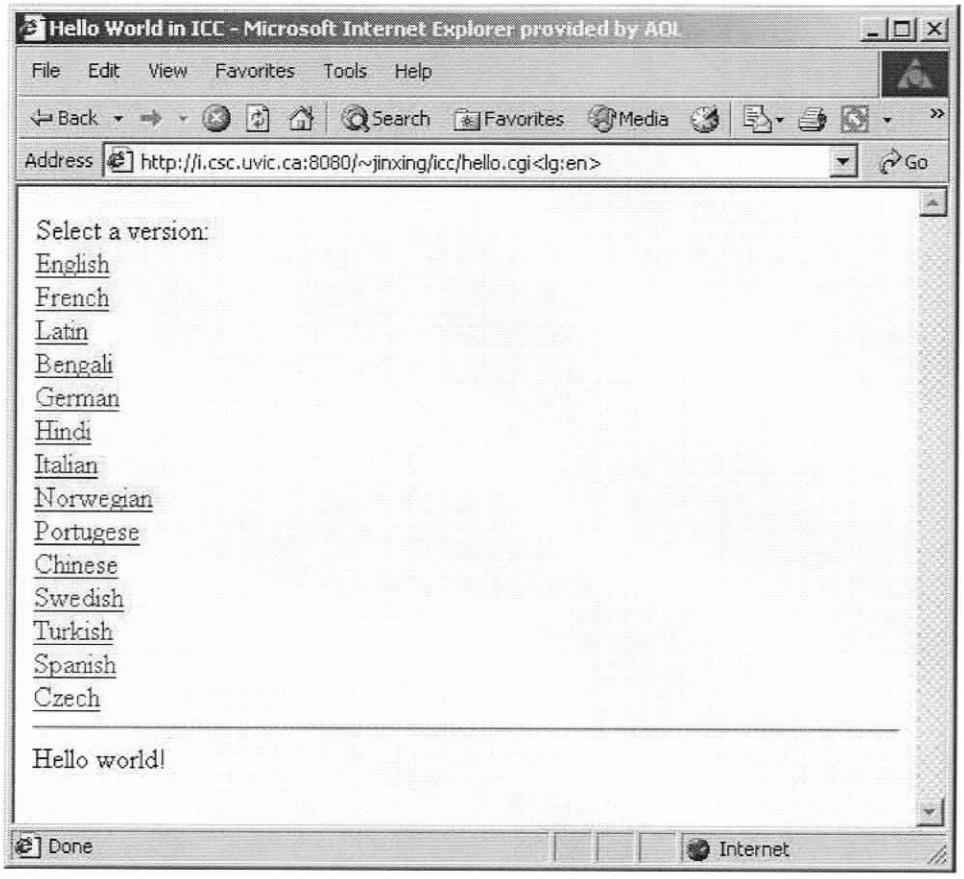


Figure B.1. *Hello World! (CGI) In ICC*

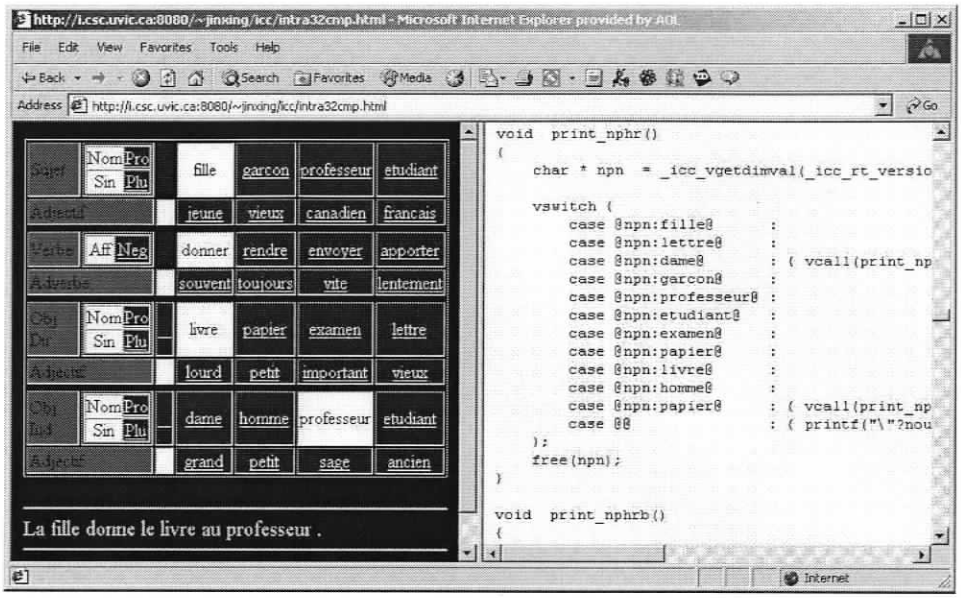


Figure B.2. *French Sentences Maker In ICC*

A more complex site, the French Sentences Maker by Bill and Christine Wadge [4], has been ported into an ICC version from its ISE version. Figure B.2 shows the screen snapshot of this ported CGI program. The left side panel shows the French Sentences Maker, while the right side panel shows the ICC source file.

The source code of the ported ICC version is almost double in the number of lines and in the size of code as that of the ISE version. This is mainly because the syntax of ISE is generally more concise and symbolized than its ICC counterpart. On the other hand, the ICC code is more readable than its ISE counterpart.

B.4.2 Migrate Legacy C Code to Intensional Code

The C programming language was devised in the early 1970's which paralleled the early development of the Unix operating system. Ever since then, the C language has spread widely, while today it is among the languages most commonly used throughout the computer industry. Volumes of applications were written in the C programming languages.

Although the trend of programming languages has seen a shift to C++ and Java in the computer industry recently, many individuals still use C as one of their routine tools. Often, softwares written in C language are required to evolve into a sub-version with only some additions and modifications of some components to the original version. One common means to address this problem is to clone the source and make some modifications on the source code with conditional compiler directives.

ICC provides another means of addressing this problem. With the subjected variables and functions changed into intensional variables and functions, respectively, to keep the old context while allowing the new context to fulfill the new requirements, the original source codes can remain almost untouched with new functionalities added in an intensional approach. The following pseudo codes are used to simplify the illustration of this approach.

Suppose a.c is the original C program with a core algorithm A in its function 'f' as shown below:

```
/* a.c */

void f() {
    /* algorithm A */
}
```

The first version of a.c is successful, but over some time an algorithm B is devised. The user requests a new version of a.c which functions exactly the same as the old version and also embeds the new functionality provided by the new algorithm B. The user then wants the new version of a.c to run under different contexts with either algorithm A or B as its core algorithm. The selection can be a command line switch or a menu option.

With ICC programming, this is an easy task as shown in the following modification to the above piece of code:

```
/* a.c - new version */

void f@@() {
    /* algorithm A */
}
```

```
void f@algorithm:B@() {
    /* algorithm B */
}
```

The above piece of code simply adds a new version of 'f' containing algorithm B and makes the old 'f' containing algorithm A as the default version for dimension 'algorithm'. In this way, the infrastructure of the old 'a.c' is kept and the new functionality is added.

Because of the huge volume of applications written in the C programming language, the application of ICC on migrating legacy C codes into intensional codes is of great interest to intensional programmers who want to place an intensional layer on top of existing C applications.